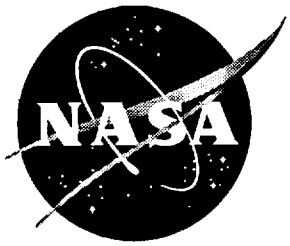
IN-62
3811

NASA Technical Memorandum 109102



# ASSIST Internals Reference Manual

Sally C. Johnson
*Langley Research Center, Hampton, Virginia*

David P. Boerschlein
*Lockheed Engineering & Sciences Company, Hampton, Virginia*

(NASA-TM-109102) ASSIST INTERNALS REFERENCE MANUAL (NASA. Langley Research Center) 101 p

N94-29494

Unclas

G3/62 0003811

April 1994

National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virginia 23681-0001

# ASSIST Internals Reference Manual

Sally C. Johnson and
David P. Boerschlein

NASA Langley Research Center
Hampton, VA 23681–5225

April 12, 1994

**Abstract**

The Abstract Semi-Markov Specification Interface to the SURE Tool (ASSIST) program was developed at NASA Langley Research Center in order to analyze the reliability of virtually any fault-tolerant system. A user manual [1] was developed to detail its use. Certain technical specifics are of no concern to the end user, yet are of importance to those who must maintain and/or verify the correctness of the tool. This document takes a detailed look into these technical issues.

# Contents


**1 Introduction** 1

**2 ASSIST Processing** 2

**3 The Pseudo Code Language Used by ASSIST** 8

3.1 Instructions . . . . . 8

3.2 Expression operators . . . . . 11

**4 Data Structures Used to Parse ASSIST Source Code** 17

4.1 The source code input line data structure . . . . . 17

4.2 The identifier table data structure . . . . . 17

4.3 The state offset data structure . . . . . 19

4.4 The token information data structure . . . . . 20

4.5 The expression data structure . . . . . 21

4.6 The SPACE statement data structure . . . . . 23

4.7 The START statement data structure . . . . . 27

4.8 The current state bit string . . . . . 28

4.9 Macro definitions and data structures . . . . . 30

4.9.1 The IMPLICIT statement data structure . . . . . 31

4.9.2 The FUNCTION statement data structure . . . . . 32

4.9.3 The macro expansion stack data structure . . . . . 33

4.10 The VARIABLE statement data structure . . . . . 39

4.11 The cross-reference-map entry data structure . . . . . 40

**5 Data Structures Used to Generate a Model File** 41

5.1 Introduction to model generation data structures . . . . . 41

5.2 The ASSERT statement data structure . . . . . 43

5.3 The DEATHIF statement data structure . . . . . 44

5.4 The PRUNEIF statement data structure . . . . . 45

5.5 The TRANTO statement data structure . . . . . 47

5.6 The VARIABLE statement data structure . . . . . 48




PRECEDING PAGE BLANK NOT FILMED

5.7 The Block IF statement data structure . . . . . 50
5.8 The FOR loop statement data structure . . . . . 51
5.9 The FOR index repetition information data structure . . . . . 55
5.10 The space expression list data structure . . . . . 58
5.11 The built-in function parameter information data structure . . . . . 61
5.12 The value union data structure . . . . . 62
5.13 The binary operand pair data structure . . . . . 63
5.14 The reserved word operator lookup data structure . . . . . 64
5.15 The reserved word lookup data structure . . . . . 64
5.16 The token lookup data structure . . . . . 65
5.17 The scanning character information data structure . . . . . 65
5.18 Mapping of a program into memory . . . . . 66

**6 Hashing of state space** **74**

**7 Concluding Remarks** **77**

**8 References** **78**

**A BNF Language Description** **79**

**B Command Line Options** **91**

**C Notes for VMS Users** **94**
C.1 Special VMS Errors . . . . . 94
C.2 Model Cannot be Piped . . . . . 94

# List of Tables

1 Object Code Table Sections . . . . . 4
2 Operand referencing operations in ALU . . . . . 11
3 Binary arithmetic operations in ALU . . . . . 12
4 Array element referencing operations in ALU . . . . . 12
5 Grouping operations in ALU . . . . . 12

6 Unary arithmetic operations in ALU . . . . . 13
7 Unary arithmetic operations in ALU . . . . . 14
8 Binary arithmetic operations in ALU . . . . . 15
9 List operations in ALU . . . . . 15
10 Concatenation operations in ALU . . . . . 15
11 Standard value push operations in ALU . . . . . 16
12 How bits are packed for the "type flagword type" . . . . . 20

# List of Figures

1 Data File Flow in ASSIST . . . . . 2
2 Format of ".aobj" file . . . . . 3
3 Format of each table entry in ".aobj" file . . . . . 3
4 Sample expression laid out in memory . . . . . 23
5 SPACE statement laid out in memory . . . . . 25
6 SPACE statement laid out in memory . . . . . 26
7 START statement laid out in memory . . . . . 28
8 Example illustrating packing of a state-space node . . . . . 29
9 Another example of packing of a state-space node . . . . . 29
10 Layout of FUNCTION definition in memory . . . . . 34
11 Overview of expansion stack during nested function invocation . . . . . 36
12 Detail of expansion stack during function invocation . . . . . 38
13 VARIABLE statement laid out in memory . . . . . 40
14 Sample ASSERT laid out in memory . . . . . 44
15 Sample DEATHIF laid out in memory . . . . . 46
16 Sample PRUNEIF laid out in memory . . . . . 47
17 TRANTO clause (list format) laid out in memory . . . . . 48
18 TRANTO clause (space expression format) laid out in memory . . . . . 49
19 VARIABLE statement laid out in memory . . . . . 50
20 Sample block IF laid out in memory . . . . . 52
21 Sample memory map of corresponding block IF . . . . . 53
22 Sample FOR laid out in memory . . . . . 56

23 Sample memory map of corresponding FOR . . . . . 57
24 Sample hash table laid out in memory . . . . . 76

# 1 Introduction

This manual is designed to be used by system administrators and programmers in order to be able to understand the internals of the ASSIST program. Parts of the manual may apply only to system administrators and other parts may apply to only programmers.

This manual is quite technical and is not intended for the typical user. Users are referred instead to the ASSIST user manual [1].

It is assumed that the reader is already somewhat familiar with the syntax and semantics of an ASSIST input file. This familiarity can be gained by reading the ASSIST user manual. It is also assumed that the reader has a basic knowledge of the ANSI "C" programming language.

After completing this manual, the reader should:

1. have an advanced understanding of the internals of ASSIST and be able to use advanced features that would be unknown to the typical end user.
2. have a detailed understanding of all data structures used by the `ASSIST` source code.
3. be able to read an `ASSIST` loadmap ( **−loadmap** or **/loadmap** option).
4. understand an `ASSIST` object file and how an `ASSIST` input description is parsed and stored in memory in preparation for model generation.
5. understand the pseudo-code language that `ASSIST` uses to generate the model file.
6. be confident about the correctness of the algorithms and data structures used to implement the language.
7. be able to maintain the source code and understand the subtle implications of any changes to it.

# 2 ASSIST Processing

The ASSIST program executes in two phases: parsing of the input file followed by model generation. As shown in Figure 1, the ASSIST program reads an input file containing the model description and creates several output files. The name of the input file must end with an *".ast"* extent. The model output file produced by the program contains the semi-Markov model. The model file has a *".mod"* file extent. The model file contains all of the named numeric (non-boolean) constants defined in the ASSIST input file as well as the model transitions. Any statements in the ASSIST input file that are surrounded by double quotes are also copied directly into the model file. The log (or listing) file contains a listing of the ASSIST input file plus various information to aid the user in checking the correctness of the model generated. The listing file has the *".alog"* file extent unless executing on systems where four character extents are illegal, in which case it has the *".alg"* file extent. Errors encountered during model generation are printed after the input file listing and optional variable and/or load maps.

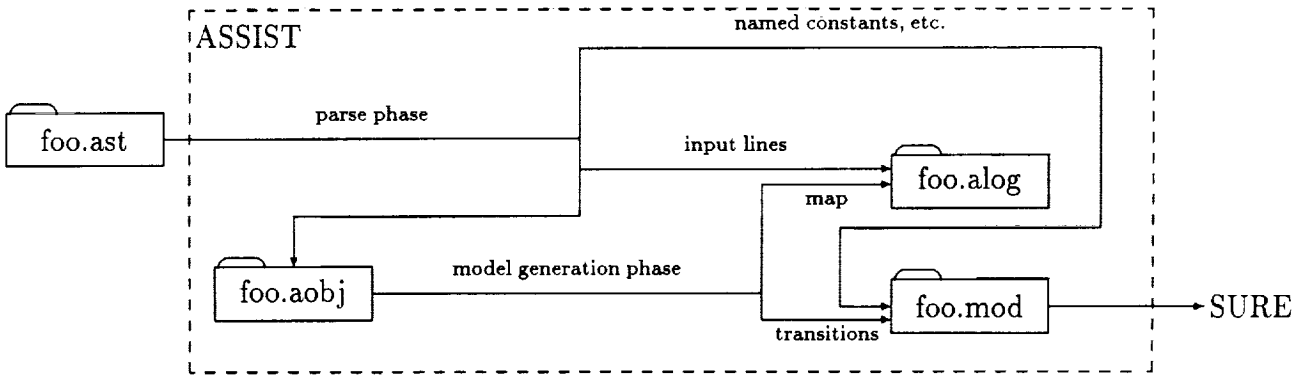



Figure 1: Data File Flow in ASSIST

The ASSIST program makes use of several temporary files. Temporary files are required to process the ASSIST `INPUT` statement and to store variable definitions for the optional cross-reference map, which is listed on the log file. Temporary files are deleted automatically after successful generation of the model file. Temporary file names are system dependent. They begin with "`QQ`" and end with a sequence of digits. They usually do not contain an extent.

During parsing of an ASSIST input file, the data and code necessary to generate the model file is written to an object file. This file is re-read and loaded into memory before generating the transitions between the states in the model. The object file has the *".aobj"* file extent unless executing on systems where four-character extents are illegal, in which case it has the *".aoj"* file extent.

The object file is defined to be a sequence of table entries. Each table entry has four fields, namely:

- section index
- count of "n" data elements
- size of each data element
- actual data ($n \times size$ bytes)

The first table entry in the object file is always the header entry and the last is always the end-of-file entry. Figures 2 and 3 give this format.

The format of the header is an array of MAX_OBJ_COUNTER_DIM elements of size memsize_t. For the VAX and SUN systems, the elements are longs. Each element is the number of bytes of memory required for each of the corresponding sections of data and/or code. These sections are listed in Table 1. Some of the constants in the table are defined in the file "objdefs.h", and others are defined in the file "astdefs.h".

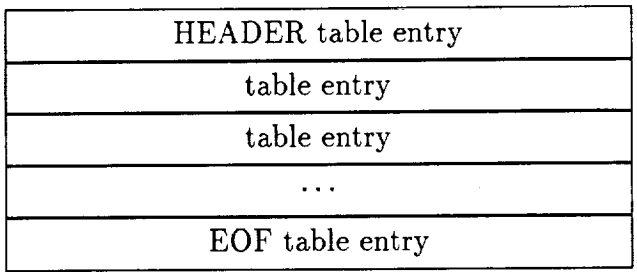



Figure 2: Format of ".aobj" file

| long | long | long | byte array |
|---|---|---|---|
| Section | count | size each | optional data |

Figure 3: Format of each table entry in ".aobj" file

When an object file is read and loaded into memory, the following steps take place in the order listed:

| Section | #define | description |
|---|---|---|
| 1 | OBJ_CHAR_DATA | Character constants |
| 2 | OBJ_BOOL_DATA | Boolean constants |
| 3 | OBJ_INT_DATA | Integer (Long) constants |
| 4 | OBJ_REAL_DATA | Real (Double) constants |
| 5 | OBJ_SOFF_DATA | State offset structure constants |
| 6 | future usage | future usage |
| 7 | future usage | future usage |
| 8 | future usage | future usage |
| 9 | OBJ_CHAR_VARDATA | Character variables |
| 10 | OBJ_BOOL_VARDATA | Boolean variables |
| 11 | OBJ_INT_VARDATA | Integer (Long) variables |
| 12 | OBJ_REAL_VARDATA | Real (Double) variables |
| 13 | OBJ_SOFF_VARDATA | State offset structure variables |
| 14 | future usage | future usage |
| 15 | future usage | future usage |
| 16 | OBJ_EXPR | Expression structures |
| 17 | OBJ_OPERANDS | Expression operand pointers |
| 18 | OBJ_OPS | Expression operation constants |
| 19 | OBJ_VARINF | Variable pointer unions |
| 20 | OBJ_SETRNGE | Set range bound structures |
| 21 | OBJ_PIX | State space picture data |
| 22 | OBJ_BOOLTEST | Boolean test expression structures |
| 23 | OBJ_TRANTO | TRANTO clause structures |
| 24 | OBJ_IF | Block if structure |
| 25 | OBJ_FOR | for construct (loop) structures |
| 26 | OBJ_CALC | for variable calculations |
| 27 | future usage | future usage |
| ⋮ | ⋮ | ⋮ |
| 39 | future usage | future usage |
| 40 | OBJ_CODE_0 | Beginning of code (PREAMBLE) |
| 41 | OBJ_CODE_0 + 1 + OPCODE_ASSERT | Code (ASSERT section) |
| 42 | OBJ_CODE_0 + 1 + OPCODE_DEATHIF | Code (DEATHIF section) |
| 43 | OBJ_CODE_0 + 1 + OPCODE_PRUNEIF | Code (PRUNEIF section) |
| 44 | OBJ_CODE_0 + 1 + OPCODE_TRANTO | Code (TRANTO section) |
| 45 | OBJ_CODE_0 + 1 + OPCODE_CALC | Code (CALC-booltest section) |
| 46 | OBJ_CODE_0 + 1 + OPCODE_CALC_T | Code (CALC-transition section) |
| 47 | future usage | Code (future section) |
| 0xf0 | OBJ_HEADER | header record |
| 0xf1 | OBJ_IDTABLE | Identifier table |
| 0xf2 | future | future use |
| 0xf3 | OBJ_OPREC | Option record |
| 0xf4 | OBJ_VERBATIM_HEAD | declarations sent to model file verbatim |
| 0xf5 | OBJ_VERBATIM_TAIL | trailing text to write to model file (such as "\nRUN;\nEXIT\n;" with **−pipe** option) |
| 0xff | OBJ_EOF | End-of-file entry |

Table 1: Object Code Table Sections

1. The object file is read and loaded in its entirety. Memory is allocated after reading the header, identifer table, and number table records (i.e, up to three blocks of memory are allocated). Memory allocated while reading the header is divided as required for the separate sections. The OBJ_OPREC data is read directly into the static storage for the "option_rec" structure. Other data is loaded into memory. The identifier table pointers are adjusted with a call to "fixup_identifier_table" upon loading the table.

2. All expression pointers are adjusted with a call to "fixup_expressions".

3. All other pointers are adjusted with a call to "fixup_user".

4. The optional memory map ( **−loadmap** ) is generated.

The model generation algorithm builds the model from the start state by recursively applying the TRANTO rules. A list of states to be processed, called the "Ready Set", begins with only the start state. Before application of a rule, ASSIST checks all of the ASSERT conditions and prints any warning messages. All death conditions are then checked to determine if the current state is a death state. Since a death state denotes system failure, no transitions can leave a death state. If the state is not a death state, ASSIST then checks all prune conditions to determine if the current state is a prune state. If ASSIST finds a state-space variable that is out of range or detects some other error in the state, the state is treated as a death state. Each of the TRANTO rules is then evaluated for the nondeath state. If the condition expression of the TRANTO rule evaluates to true for the current state, then the destination expression is used to determine the state-space variable values of the destination state. If the destination state has not already been defined in the model, then the new state is added to the Ready Set of states to be processed. The rate of the transition is determined from the rate expression, and the transition description is printed to the model file. When all of the TRANTO rules have been applied to it, the state is removed from the Ready Set. When the Ready Set is empty, then all possible paths terminate in death states, and model building is complete.

By default, all death states are aggregated or lumped (ONEDEATH ON) according to the first DEATHIF statement to which the state conformed. If the user sets ONEDEATH OFF, then all distinct death states are kept in the model.

Note that the ready list is a subset of the set of all state nodes that have been processed up to any given point in time. All state nodes that have been processed must remain in memory because ASSIST must check each new destination state to see if it has already been processed. There are typically many different paths to each state in the model. States that have already been processed are not processed again.

The following is a pseudo-code version of the algorithm used to generate the model:

```
(* ====================== subroutines/functions ==================== *)
FUNC  PROCESS(state,trim,fast,in_error)
      (* note that ''fast'' is ignored when trimming is off *)):
      state number ⟵ search existing states.
      IF (state already present) THEN:
         is a death state if flagged as such.
```

```
      ELSE:
         save current state and dependent variable values.
            recompute dependent variables that are referenced in
                 ASSERT, DEATHIF, PRUNEIF sections for state.
            test all ASSERT's, printing WARNING message if a test fails.
            test all DEATHIF's.
            IF (not a death) test all PRUNEIF's.
         restore current state and dependent variable values.
         IF (death or prune state) AND (fast) AND (trimming is on) THEN:
            print warning message:
                 Model contains recovery transitions directly to death state
                 and therefore may not be suited to trimming.
         ENDIF.
      ENDIF.
      IF (death state) AND (lumping) THEN:
         state number ⟵ death state number.
      ELSE IF (prune state) THEN:
         state number ⟵ prune state number.
      ELSE IF ((trim) AND (trimming is on)) THEN:
         state number ⟵ trim state number.
      ELSE (* not being lumped *):
         IF (death state) THEN:
            flag the state.
         ENDIF.
         IF (state does not yet exist) THEN:
            state number ⟵ add the state to the ready list.
         ENDIF.
      ENDIF.
      RETURN state number.
ENDFUNC.

(* ========================= main algorithm ========================= *)

MAIN:
    (* process the start state *)
    compute start state.
%    compute dependent variables referenced in TRANTO section.
    start state number ⟵ call PROCESS(start state,NORMAL,N/A,error).

    (* generate the model *)
    ready list ⟵ pointer to first state.
    FOR current-state IN [all states on ready list] LOOP:
        IF (state is not flagged as death state) THEN:
           set fast transition counter to zero.
           FOR (all recovery (fast) TRANTO's) DO:
       compute new state.
              new state number ⟵ call PROCESS(new state,NORMAL,FAST,error).
              print the transition to the model file.
              increment fast transition counter.
           ENDFOR.
           IF (trimming is on) THEN:
               set slow transition counter to zero.
               FOR (all non-recovery (slow) TRANTO's) DO:
                   compute new state.
                   IF (fast transition counter > 0) THEN:
                      new state number ⟵ call PROCESS(new state,TRIM,SLOW,error).
                   ELSE:
                      new state number ⟵ call PROCESS(new state,NORMAL,SLOW,error).
                   ENDIF.
                   print the transition to the model file.
               ENDFOR.
           ELSE:
               set slow transition counter to zero.
```

```
                FOR (all non-recovery (slow) TRANTO's) DO:
                    compute new state.
                    new state number ⟵ call PROCESS(new state,NORMAL,N/A).
                    print the transition to the model file.
                ENDFOR.
            ENDIF.
         ENDIF.
         print warning if no transitions out of a non-death state.
         ready list ⟵ increment pointer to next ready state.
      ENDFOR.

      (* print extra trim transitions *)
      IF (trimming is on) THEN:
         FOR current-state IN [trim state only] DO:
             print transition from current-state to trim death
                 state BY TRIMOMEGA.
         ENDFOR.
         IF  (pruning with TRIMOMEGA (i.e., trim=2)) THEN
             FOR current-state IN [prune states] DO:
                 print transition from current-state to current prune
                       death state BY TRIMOMEGA.
             ENDFOR.
         ENDIF.
      ENDIF.
      FOR (all TRANTO's) DO:
         IF (never referenced) THEN:
             print a warning message that TRANTO was never used.
         ENDIF.
      ENDFOR.
      IF (fatal error occurred) THEN:
         flag model file as erroneous.
      ENDIF.
STOP.
```

# 3 The Pseudo Code Language Used by ASSIST

The pseudo code language used by ASSIST is similar to a stripped down assembly language with instructions that closely resemble statements and expression operations that closely resemble expression syntax in the ASSIST language.

## 3.1 Instructions

Every instruction has two parts, namely an operation code and a pointer to either a data structure or another instruction. The instruction pointer union occurs first in the structure because some system architectures require pointer alignment on full or half word boundaries. The opcode comes first in the pseudo language, so it will be described first.

An instruction is stored in data of the following types:

```
typedef struct t__instruction_pointer_union
   {
       void *vvv;                          /* to cast to block_if_type, etc. */
       relative_address_type reladdr;      /* relative address of code */
   } instruction_pointer_union_type;

typedef struct t__instruction
   {
       instruction_pointer_union_type ptr;
       opcode_type opcode;          /* instruction operation code */
   } instruction_type;              /*    for_loop_type, assert_type, ... */
```

The following operation codes are defined:

- The **ASSERT** instruction makes an assertion and prints a warning message when the current state does not pass the assertion. It has one required parameter, which is a non-null pointer to a Boolean test data structure for the condition to be tested for conformance. See Section 5.2 on page 43 for details on this data structure.
- The **DEATHIF** instruction tests the current state and signals system failure when a condition is met. It has one required parameter, which is a non-null pointer to a Boolean test data structure for the condition to be tested for system failure. See Section 5.3 on page 44 for details on this data structure.
- The **PRUNEIF** instruction tests the current state and signals system failure due to model pruning when a condition is met. It has one required parameter, which is a non-null pointer to a Boolean test data structure for the condition to be tested for system failure due to model pruning. See Section 5.4 on page 45 for details on this data structure.
- The **TRANTO** instruction defines a transition from one state to another state. It has one required parameter, which is a non-null pointer to a transition data structure. See Section 5.5 on page 47 for details on this data structure.

- The **CALC** instruction defines the computation of a dependent variable. Such variables differ from named constants in that they are dependent upon the state-space variables. See Section 5.6 on page 48 for details on this data structure.

- The **START** instruction defines a transition to the start state. It has one required parameter, which is a non-null pointer to a transition data structure. The starting transition occurs in the preamble section code and points to a transition containing all constant expressions with no references to any variables. See Sections 4.7 and 5.5 on pages 27 and 47 for details on this data structure.

- The **SPACE** instruction occurs in the preamble and points to the state-space picture. It causes an address to be loaded into the state-space picture register. The state-space picture is used when a comment is printed to the model file. The SPACE instruction has one required parameter, which is a non-null pointer to the state-space picture data. See Section 4.6 on page 23 for details on this data structure.

- The **BLOCK_IF** instruction can occur in any section except the preamble section. It has one required parameter, which is a non-null pointer to a block IF data structure as defined later in Section 5.7. The data structure must contain test condition data and pointers to THEN and ELSE subroutines. The IF pseudo-instruction will set aside space for a data structure and load it with the addresses for the GOSUB instructions. For example, "IF <expr> THEN GOSUB <code>" or "IF <expr> THEN GOSUB <code> ELSE GOSUB <code>". See Section 5.7 on page 50 for details on this data structure.

- The **FOR_LOOP** instruction can occur in any section except the preamble section. It has one required parameter, which is a non-null pointer to a FOR data structure as defined later in Section 5.8. See Section 5.8 on page 51 for details on this data structure.

- The **BEGIN** instruction denotes the beginning of a rule section of code. There must appear in the preamble section exactly one BEGIN instruction for each of the rule sections. These must appear in the correct sequence. Each BEGIN instruction has one required parameter, which is a non-null pointer to the beginning of the code for the corresponding rule section. The first BEGIN corresponds to the first rule section (ASSERT section), the second to the second section (DEATHIF section), etc. When a program is executed, the BEGIN addresses are stored in the BEGIN registers and an implicit GOSUB is executed to each of these addresses for each state in the model during processing (see Section ??).

- The **END** instruction is used to terminate the preamble section of the code. It has no parameters.

- The **GOTO** instruction is used to jump to an instruction. It has one required parameter, which is a non-null pointer to the instruction where processing is to continue. Control will never continue with the next instruction following a GOTO unless there is another GOTO that points to it.

- The **GOSUB** instruction denotes transfer of control to a subroutine. It causes a recursive call to the subroutine evaluator unit, which processes instructions until either an END or RETURN instruction is encountered.
- The **RETURN** instruction denotes the end of a subroutine and instructs the subroutine evaluator to return to its invoking activation level causing processing to resume with the instruction immediately following the most recently encountered GOSUB. If the activation level was invoked due to a BEGIN instruction, then processing of the current state will continue on as detailed in Section ??.

The instruction pointer union contains a pointer to another instruction or one of the following data types:

```
typedef struct t__booltest
   {
       expression_type *expr;          /* boolean expr to ASSERT,DEATHIF,etc. */
       short source_code_line_number; /* line number in listing file */
       short lumping_sequence;         /* sequence index (0..n-1) in source */
   } booltest_type;                    /* e.g., first DEATHIF, second DEATHIF */

typedef struct t__block_if
   {
       expression_type *then_test;     /* boolean expression for THEN */
       instruction_type *then_clause; /* code for THEN clause */
       instruction_type *else_clause; /* code for ELSE clause */
   } block_if_type;

typedef struct t__for_loop
   {
       identifier_info_type *ident;  /* index variable */
       set_range_type *set_ranges;   /* pointer to array of IN ranges */
       short set_range_count;        /* count of number of IN ranges */
       instruction_type *body;       /* pointer to BODY of loop */
   } for_loop_type;

typedef struct t__tranto_clause
   {
       space_expression_type sex;   /* list of space transition expressions */
       expression_type *rate_exprs; /* ptr to array of rate expressions */
       short n_rate_exprs;          /* count of rate expressions */
       short source_code_line_number;
   } tranto_clause_type;

typedef struct t__state_space_picture
   {
       vars_union_type *varu;
       Boolean *is_nested;
       short nvaru;
   } state_space_picture_type;
```

The kind of data structure pointed to is inherently implied by the specific operation code in the instruction.

The language is designed so as to guarantee that the first "n" opcodes correspond to the rule sections and are numbered 0 through $n-1$. All opcodes are ordinally contiguous. The value of "n" is given by the RULE_OPCODE_INDEX_COUNT constant.

The section in which an opcode appears is masked into the high bits of the opcode field.

Examples of pseudo code programming are not given here because pseudo code programs are not directly written by ASSIST users. If the user asks for a load map ( **−loadmap** option), the format of the map will use these pseudo operation codes. See Section 5.18 on page 66 for a sample memory layout, which includes the pseudo code.

## 3.2 Expression operators

Expressions are made up of operands, which are pointers into the identifier table, combined with operations for the Arithmetic/Logic Unit (ALU).

The same set of operations are used in both infix and postfix expressions. A few of the operations in the set, however, apply only to one or the other.

There are two operations that are used to instruct the unit to look for an operand. These are listed in Table 2. Both of these operations are of the highest precedence.

| | |
|---|---|
| OP_VAL | Operand specified by user |
| OP_INSVAL | Operand inserted during parsing |

Table 2: Operand referencing operations in ALU

The remaining operations will be covered from lowest precedence to highest precedence.

The binary arithmetic operations act upon two numerical values and yield either a numeric or a logical (Boolean) result. These operations are given in Table 3.

Certain arithmetic operations act upon a pointer to the beginning of an array and the integer value of an index. The pointer actually points into the identifier table so that type and range bound information is available. These operations are given in Table 4. Note that `OP_IXDBY2` is a tertiary operator since it acts upon a pointer into the identifier table followed by a primary index value followed by a secondary index value.

Certain additional operations are used in infix expressions in order to group operands and/or expressions together. These are listed in Table 5.

Unary arithmetic operators are given in Table 6.

Unary built-in function operators are additional unary operators that are included in the ALU in order to more efficiently process built-in functions in ASSIST. These are listed in Table 7.

Binary built-in function operations are additional binary operators that are included in the ALU in order to more efficiently process built-in functions in ASSIST. These are listed

| | |
|---|---|
| OP_OR | $x$ OR $y$ |
| OP_XOR | $x$ XOR $y$ |
| OP_AND | $x$ AND $y$ |
| OP_BOOL_EQ | $x == y$ |
| OP_BOOL_NE | $x \sim\sim y$ |
| OP_LT | $x < y$ |
| OP_GT | $x > y$ |
| OP_LE | $x <= y$ |
| OP_GE | $x >= y$ |
| OP_EQ | $x = y$ |
| OP_NE | $x <> y$ |
| OP_ADD | $x + y$ |
| OP_SUB | $x - y$ |
| OP_MUL | $x * y$ |
| OP_DVD | $x / y$ |
| OP_MOD | $x$ MOD $y$ |
| OP_CYC | $x$ CYC $y$ |
| OP_QUO | $x$ DIV $y$ |
| OP_POW | $x ** y$ (infix only) |
| OP_RPOWR | $x ** y$ (postfix only) |
| OP_IPOWI | $i ** n$ (postfix only) |
| OP_RPOWI | $x ** n$ (postfix only) |

Table 3: Binary arithmetic operations in ALU

| | |
|---|---|
| OP_IXDBY | $v$ [ $i$ ](postfix only) |
| OP_IXDBY2 | $v$ [ $i, j$ ](postfix only) |
| OP_I_LB | [(infix only) |
| OP_I_WILD | * (infix only) |
| OP_I_RB | ](infix only) |

Table 4: Array element referencing operations in ALU

| | |
|---|---|
| OP_PARENS | ( ) (future use) |
| OP_I_LP | ( (infix only) |
| OP_I_RP | ) (infix only) |
| OP_I_CMMA | , (infix only) |

Table 5: Grouping operations in ALU

| | |
|---|---|
| OP_INC | $n\,{+}{+}$ |
| OP_DEC | $n\,{-}{-}$ |
| OP_NOT | NOT $p$ |
| OP_NEG | $-\,x$ |
| OP_STNCHR | $x \longrightarrow blank$ |
| OP_STNBOO | $x \longrightarrow$ FALSE |
| OP_STNINT | $n \longrightarrow 0$ |
| OP_STNRE | $x \longrightarrow 0.0$ |
| OP_ItoR | $n \longrightarrow n.000000$ |
| OP_BtoI | $p \longrightarrow \begin{cases} 0 \text{ if } \not p \\ 1 \text{ if } p \end{cases}$ |

Table 6: Unary arithmetic operations in ALU

in Table 8. Note that the wildcard row/column operations are binary because, in postfix, they act upon a pointer into the identifier table followed by the value of the column or row subscript, which will remain fixed during the summation.

There are some list functions that take an undetermined number of parameters. The postfix notation for these functions is a list of parameters followed by a count of the number of parameters followed by the list operator itself. For example:

```
SUM(6,ARRAY,18)
```

would be represented as the postfix:

```
Operands:  6,ARRAY,18,3
Operators: OP_VAL,OP_VAL,OP_VAL,OP_INSVAL,OP_ILISSUM
```

The list function operators are listed in Table 9.

The binary variable concatenation construction operation is listed in Table 10.

There are also some operations that push standard values such as zero and one onto the evaluation stack. These operations are faster than having to look up the value in memory. These are listed in Table 11.

| | |
|---|---|
| OP_SQRT | SQRT($x$) |
| OP_EXP | EXP($x$) |
| OP_LN | LN($x$) |
| OP_ABS | ABS($x$) |
| OP_SIN | SIN($x$) |
| OP_COS | COS($x$) |
| OP_TAN | TAN($x$) |
| OP_ARCSIN | ARCSIN($x$) |
| OP_ARCCOS | ARCCOS($x$) |
| OP_ARCTAN | ARCTAN($x$) |
| OP_FACT | FACT($x$) |
| OP_GAM | GAM($x$) |
| OP_SIZE | SIZE($arr$) |
| OP_COUNT1 | COUNT1($x$) (identity function) |
| OP_IMIN1 | IMIN1($x$) (identity function) |
| OP_RMIN1 | RMIN1($x$) (identity function) |
| OP_IMAX1 | IMAX1($x$) (identity function) |
| OP_RMAX1 | RMAX1($x$) (identity function) |
| OP_ISUM1 | ISUM1($x$) (identity function) |
| OP_RSUM1 | RSUM1($x$) (identity function) |
| OP_ANY1 | ANY1($x$) (identity function) |
| OP_ALL1 | ALL1($x$) (identity function) |
| OP_COUNT | COUNT($arr$) |
| OP_IMIN | IMIN($arr$) |
| OP_RMIN | RMIN($arr$) |
| OP_IMAX | IMAX($arr$) |
| OP_RMAX | RMAX($arr$) |
| OP_ISUM | ISUM($arr$) |
| OP_RSUM | RSUM($arr$) |
| OP_ANY | ANY($arr$) |
| OP_ALL | ALL($arr$) |

Table 7: Unary arithmetic operations in ALU

| | |
|---|---|
| OP_COMB | COMB($n$,$k$) |
| OP_PERM | PERM($n$,$k$) |
| OP_ROWCOUNT | COUNT($arr[i,*]$) |
| OP_COLCOUNT | COUNT($arr[*,i]$) |
| OP_IROWMIN | MIN($intarr[i,*]$) |
| OP_ICOLMIN | MIN($intarr[*,i]$) |
| OP_RROWMIN | MIN($realarr[i,*]$) |
| OP_RCOLMIN | MIN($realarr[*,i]$) |
| OP_IROWMAX | MAX($intarr[i,*]$) |
| OP_ICOLMAX | MAX($intarr[*,i]$) |
| OP_RROWMAX | MAX($realarr[i,*]$) |
| OP_RCOLMAX | MAX($realarr[*,i]$) |
| OP_IROWSUM | SUM($intarr[i,*]$) |
| OP_ICOLSUM | SUM($intarr[*,i]$) |
| OP_RROWSUM | SUM($realarr[i,*]$) |
| OP_RCOLSUM | SUM($realarr[*,i]$) |
| OP_ROWANY | ANY($boolarr[i,*]$) |
| OP_COLANY | ANY($boolarr[*,i]$) |
| OP_ROWALL | ALL($boolarr[i,*]$) |
| OP_COLALL | ALL($boolarr[*,i]$) |

Table 8: Binary arithmetic operations in ALU

| | |
|---|---|
| OP_LISCOUNT | COUNT($p_1$,$p_2$,...,$p_n$) |
| OP_ILISMIN | MIN($i_1$,$i_2$,...,$i_n$) |
| OP_RLISMIN | MIN($x_1$,$x_2$,...,$x_n$) |
| OP_ILISMAX | MAX($i_1$,$i_2$,...,$i_n$) |
| OP_RLISMAX | MAX($x_1$,$x_2$,...,$x_n$) |
| OP_ILISSUM | SUM($i_1$,$i_2$,...,$i_n$) |
| OP_RLISSUM | SUM($x_1$,$x_2$,...,$x_n$) |
| OP_LISANY | ANY($p_1$,$p_2$,...,$p_n$) |
| OP_LISALL | ALL($p_1$,$p_2$,...,$p_n$) |

Table 9: List operations in ALU

| | |
|---|---|
| OP_CONCAT | $x \wedge n$ |

Table 10: Concatenation operations in ALU

| | |
|---|---|
| OP_PZ | Push binary zeros (0,FALSE) |
| OP_PRZ | Push real zero (0.0000000000) |
| OP_PU | Push integer unity (1) |
| OP_PBU | Push Boolean unity (TRUE) |
| OP_PRU | Push real unity (1.0000000000) |
| OP_NIX | No operation |

Table 11: Standard value push operations in ALU

# 4 Data Structures Used to Parse ASSIST Source Code

In order to understand the internals of ASSIST, it is necessary to understand the data structures used by the program and the rationale behind how they are set up. It is the purpose of this chapter to provide a basis for this understanding.

The sections in this chapter are organized topically instead of alphabetically. Some sections therefore reference more than one data structure and others may not reference any data structures. All data structures used by ASSIST are referenced at least once. Some data structures are referenced more than once.

## 4.1 The source code input line data structure

Each source code input line is stored in a structure of the following type:

```
typedef struct t__source_line_info
   {
        short last_line_in_error;
        short last_line_in_warning;
        short error_count_this_line;
        short warning_count_this_line;
        short old_line_number;
        short line_number;
        Boolean line_shown_on_screen;
        scanning_character_info_type char_pair;
        char old_line_buffer[LINE_MAXNCH_P];
        char line_buffer[LINE_MAXNCH_P];
        short line_buffer_ix;
        Boolean must_fudge_it;
   } source_line_info_type;
```

## 4.2 The identifier table data structure

The identifier table is stored as an array of elements of the following type:

```
typedef struct t__identifier_info
  {
     pointer_union_type ptr;    /* address in memory / function-parm-count */
     union
        {
           struct qqbothidinfqq
              {
                 dim_pair_type first;
                 dim_pair_type second;
              } dims;
           dim_pair_type body;
        } index;
     short scope_level;         /* scope level (negative iff. inactive) */
     char name[IDENT_MAXNCH_P]; /* identifier to search for */
     type_flagword_type flags;  /* type information */
  } identifier_info_type;
```

where the data structure for a dimension pair is of the following type:

```
typedef struct t__dim_pair
  {
    Subscript lower;/* lower dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
    Subscript upper;/* upper dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
  } dim_pair_type;
```

and where the data structure for a pointer union is of the following type:

```
typedef union t__pointer_union
   {
     relative_address_type relative_address;
     short parameter_count;
     void *vvv;                /* included for completeness */
     Boolean *bbb;             /* used when BOOL_TYPE */
     state_offset_type *sss;   /* used when SSVAR_TYPE */
     char *ccc;                /* used when CHAR_TYPE */
     int_type *iii;            /* used when INT_TYPE */
     real_type *rrr;           /* used when REAL_TYPE */
   } pointer_union_type;
```

The "ptr" field points to the data in memory for the identifier in question. If the identifier corresponds to a state-space variable, then the data is an offset into the current state string and a bit-string length.

The "scope_level" field is used when parsing FOR and block IF structures to ascertain that the same variable name is not used for nested structures and to ascertain that the scope of a variable is still active. It is also used to keep track of formal parameters in macro definition bodies.

For example, the following is *illegal* because the scope of III would still be active when it was being re-defined in the nested FOR loop:

```
FOR III IN [1..10]
    FOR III IN [1..2]
    ENDFOR
ENDFOR
```

Also, the following would be *illegal* because the scope of III is no longer active after the ENDFOR has terminated the loop.

```
FOR III IN [1..10]
    ...
ENDFOR
IF (III>7) ...
```

The "index" field is used for the subscript bound(s) if the identifier is for an array constant or an array state-space variable. It points to the beginning of the body of a macro (IMPLICIT variables and FUNCTION) definition when the identifier is for the name of a function. For arrays, both the "dims.first" and "dims.second" sub-fields are used for the first and second

doubly subscripted array bounds, respectively. If the values stored in a bound are equal to the special "SIMPLE_IDENTIFIER" value, then that bound does not apply. For macros, only the "body" sub-field applies. Note that the "dims.first" and "dims.second" sub-fields are sequential in memory whereas the "body" subfield overlays the "dims" subfield. This is done to save memory because an identifier will either be an array or a macro but never both.

Note that an IMPLICIT array is implemented as a macro, not as an array. Bounds do not apply since substituted parameters are not checked for syntactical/semantic correctness until actually parsed. The count of the number of parameters expected for a macro is stored in the "ptr.parameter_count" field.

The "lower" sub-field of either the "dims.first" or "dims.second" field is used for the lower subscript bound if the identifier is for an array constant or an array state-space variable.

The "lower" sub-field of the "body" field points to the first token for the body of a FUNCTION or IMPLICIT definition when the identifier is for the name of a macro.

The "upper" sub-field of either the "dims.first" or "dims.second" field is used for the upper subscript bound if the identifier is for an array constant or an array state-space variable.

The "upper" sub-field of the "body" field points one token past the end of the body of a FUNCTION or IMPLICIT definition when the identifier is for the name of a macro. This may or may not point to a valid token. The test "token-pointer < upper" on a while loop will guarantee that the correct tokens for a body will be accessed.

The "name" field is used to store the name of the identifier. This field is 32 characters long with 4 characters reserved for the implementation and 28 characters reserved for the user. Identifier names can therefore be at most 28 characters long. Literal number strings are also stored in the identifier name field so that they can be easily printed when printing the rate expressions to the model file. If an identifier is a literal number string, it will begin with a pound sign (#). A digit string can be at most 28 characters long. The number "-6.023000000000000000000E+23" is therefore legal but "-6.0230000000000000000000E+23" is illegal because it is one digit too long. Since "real_type" is defined to be "double" when floats are only 32 bits long and is defined to be "float" when floats are 60 or 64 bits long, the precision of the machine allows for only about 12-18 significant digits at most, so this restriction of 20 decimal places in scientific notation should not pose any serious limitations.

The "flags" field is a string of 8 bits, which are packed as shown in Table 4.2.

Unless at least one of the variable bits (bits 3,4,5,6) is set, the identifier will correspond to a constant (either a named constant or a literal value).

Several examples of how identifiers are laid out in memory are incorporated into Figure 5 on page 25 and into Figure 6 on page 26.

## 4.3 The state offset data structure

Current and new state offset information is stored in data of the following type:

| bit(s) | interpretation |
|---|---|
| 0-2 | computational type (char,bool,int,real, or state-offset) |
| 3 | expression result variable or IMPLICIT variable |
| 4 | FOR loop index variable |
| 5 | state-space variable |
| 6 | FUNCTION (IMPLICIT variable if bit 3 is also set) |
| 7 | array constant or array variable |

Table 12: How bits are packed for the "type flagword type"

```
typedef short ssvar_value_type;
typedef struct t__state_offset
   {
      ssvar_value_type minval;
      ssvar_value_type maxval;
      bitsize_type bit_offset;
      bitsize_type bit_length;
   } state_offset_type;
```

Two bit strings are maintained during rule generation, namely the source and destination states. The offset and length of a particular state are used to locate the value for a specific state-space variable within the state in question.

The "minval" field specifies the minimum value that a state-space variable is allowed to hold.

The "maxval" field specifies the maximum value that a state-space variable is allowed to hold. The difference "maxval – minval" can be at most 255.

The "bit_offset" field specifies the offset into the bit string where the packing/unpacking of a state-space variable begins.

The "bit_length" field specifies the length of the packed state-space variable in number of bits. The length can be at most 8. The value actually packed into the space is not the value itself but rather the difference "actualvalue – minval".

Several examples of how state offsets are laid out in memory are incorporated into Figure 5 on page 25 and into Figure 6 on page 26.

## 4.4 The token information data structure

The lexical scanner translates an input text file into a sequence of tokens. Each token is stored in an element of the following data type:

```
typedef struct t__token_info

      identifier_info_type *id_info_ptr;
      short linnum;
      short pos;
```

```
    token tok;
    rwtype rw;
    char id[IDENT_MAXNCH_P];
token_info_type;
```

The "id_info_ptr" field is a pointer to the corresponding entry in the identifier table.

The "linnum" field contains the line number on which the token occurred.

The "pos" field contains the column number (indexed beginning with zero) at which the first character of the token appears.

The "tok" field contains the token itself. Examples of tokens are "TK_RW" for a reserved word, "TK_ID" for an identifier, "TK_REAL" for a literal real value, "TK_SUB" for the subtraction "−" token, etc. Tokens are defined in the "tokdefs.h" file.

The "rw" field contains "RW_NULL" unless the "tok" field is "TK_RW" in which case it contains the value corresponding to the reserved word. Examples are "RW_TRANTO", "RW_START", "RW_BY", "RW_IN", etc. Reserved words are defined in the "rwdefs.h" file.

The "id" field contains the characters string for the token itself. If the token is a formal macro parameter, then this string begins with a dollar ($) sign followed by the encoded formal parameter number (beginning with one). If the token is a literal value, then this string begins with a pound (#) sign followed by the character string for the number as it was typed into the input file. If the token is a literal character string value (as in an INPUT prompt message), then this string contains "#"". If the token is an identifier, then this string contains the name of the identifier.

Several examples of tokens laid out in memory are incorporated into Figure 10 on page 34.

## 4.5 The expression data structure

Expressions occur in the syntax of many different statements and clauses in the ASSIST language. Among these are the `SPACE`, `START`, `ASSERT`, `DEATHIF`, `PRUNEIF`, `IF`, `FOR`, and constant definition statements.

Expression results can be either whole, real, or Boolean. The same data structures are used regardless of the evaluation type of the expression. The following data structures are used:

```
typedef struct t__expression
   {
       operation_type *postfix_ops;
       operation_type *infix_ops;
       operand_type *operands;
       short n_postfix_ops;
       short n_infix_ops;
       short n_operands;
       short source_code_line_number;
       Boolean in_error;
       type_flagword_type rtntype;
   } expression_type;
```

The "postfix_ops" field is a pointer to an array of postfix operations and operators. If an element in the array is an OP_VAL or an OP_INSVAL, denoted as "V" and "_V_", respectively, then a value is taken from the operand array; otherwise the element indicates either an arithmetic or Boolean operation.

The "infix_ops" field is a pointer to an array of infix operations and operators. It is similar to "postfix_ops" except that it is used for printing expressions instead of evaluating them.

The "operands" field is a pointer to an array of operands. Each operand is a pointer into the identifier table.

The "n_postfix_ops" field specifies the length of the "postfix_ops" array.

The "n_infix_ops" field specifies the length of the "infix_ops" array.

The "n_operands" field specifies the length of the "operands" array.

The "source_code_line_number" indicates the line number in the source code (*".ast"*) file as listed in the log (*".alog"*) file where the expression began. It is used to print intelligent error messages during both parsing and model generation phases.

The "in_error;" field indicates that an error was detected while attempting to parse an expression. An attempt to evaluate an expression that is "in_error" may result in a core dump or a fatal traceback error. The expression evaluator therefore returns the default value when attempting to evaluate an expression in error.

The "rtntype" field indicates the type flags for the return value of an expression.

As an example, consider the following expression:

```
(00174): ... NELE + ( 4 - 2*NIX ) / 1.0 ** MU .....
```

Figure 4 illustrates how the above expression will be stored in memory, where "I→R" stands for an explicit integer-to-real conversion and where "R**R" stands for a real number raised to a real power.

When a postfix expression is evaluated, the next value is taken from the "operands" list whenever an OP_VAL or an OP_INSVAL postfix operation is encountered. These two operations are used to instruct the evaluation that an operand comes next. Any other symbol is consequently interpreted as either an arithmetic or Boolean operation.

The lists work in a similar manner when an infix expression is printed.

The postfix list is used when an expression is evaluated. Expressions are evaluated in START and FOR statements. Boolean expressions are evaluated during rule generation whenever a test (such as a DEATHIF, IF, ASSERT, etc.) is made.

The infix list is used when an expression is printed. Expressions are printed if the user asks for a map with the VMS **/MAP** or the UNIX **−MAP** command line option. Expressions are also printed for certain "DEBUG$" options, such as "DEBUG$ PARSE$". Rate expressions are printed to the model file.

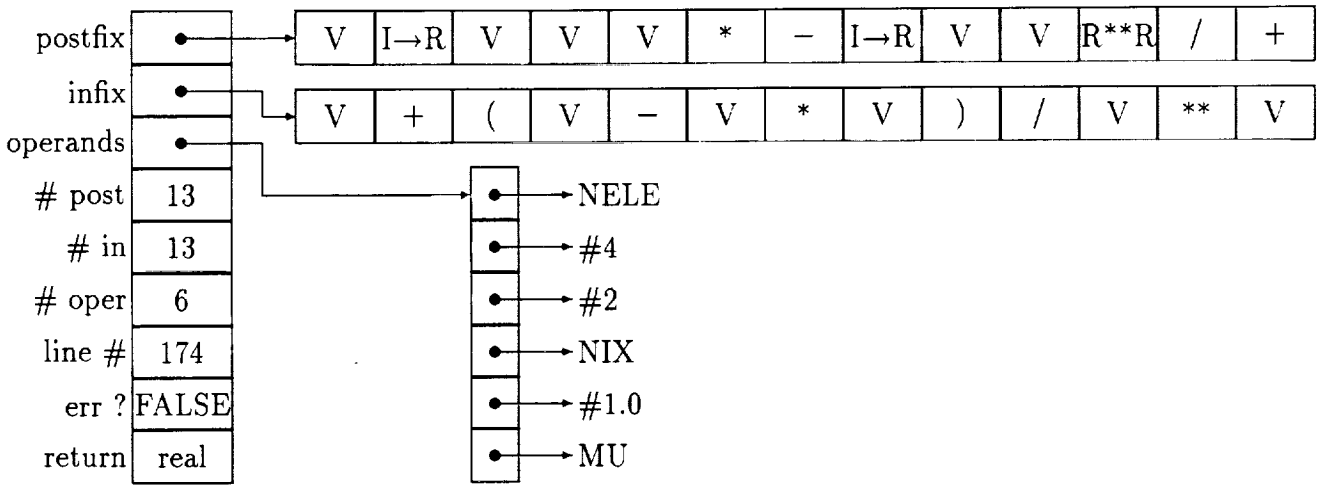



Figure 4: Sample expression laid out in memory

## 4.6 The SPACE statement data structure

The SPACE statement is parsed and stored in data structures of the following types:

```
typedef struct t__state_space_picture
   {
       vars_union_type *varu;
       Boolean *is_nested;
       short nvaru;
   } state_space_picture_type;

typedef union t__vars_union
   {
       identifier_info_type *id_info;
       state_space_picture_type *nested_space_picture;
       relative_address_type relative_address;
   } vars_union_type;

typedef struct t__identifier_info
  {
     pointer_union_type ptr;     /* address in memory / function-parm-count */
     union
        {
           struct qqbothidinfqq
              {
                 dim_pair_type first;
                 dim_pair_type second;
              } dims;
           dim_pair_type body;
        } index;
```

```
    short scope_level;          /* scope level (negative iff. inactive) */
    char name[IDENT_MAXNCH_P]; /* identifier to search for */
    type_flagword_type flags;  /* type information */
  } identifier_info_type;
```

where the data structure for a dimension pair is of the following type:

```
typedef struct t__dim_pair
  {
    Subscript lower;/* lower dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
    Subscript upper;/* upper dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
  } dim_pair_type;
```

and where the data structure for a pointer union is of the following type:

```
typedef union t__pointer_union
   {
     relative_address_type relative_address;
     short parameter_count;
     void *vvv;                 /* included for completeness */
     Boolean *bbb;              /* used when BOOL_TYPE */
     state_offset_type *sss;    /* used when SSVAR_TYPE */
     char *ccc;                 /* used when CHAR_TYPE */
     int_type *iii;             /* used when INT_TYPE */
     real_type *rrr;            /* used when REAL_TYPE */
   } pointer_union_type;

typedef short ssvar_value_type;
typedef struct t__state_offset
   {
      ssvar_value_type minval;
      ssvar_value_type maxval;
      bitsize_type bit_offset;
      bitsize_type bit_length;
   } state_offset_type;
```

The "varu" field is a pointer to an array of pointers. Each pointer in the array of pointers points to either an element in the identifier table or a nested state-space picture, depending upon whether the "is_nested" element is **FALSE** or **TRUE**, respectively.

The "is_nested" field is a pointer to an array of Booleans, which indicate whether the corresponding position in the picture is a variable or a nested state-space picture.

The "nvaru" field gives a count of the number of items in the "varu" array, which is the same as the number of items in the "is_nested" array.

The "id_info" field of the pointer union is used when "is_nested" is FALSE. The "nested_space_picture" field of the pointer union is used when "is_nested" is TRUE. The "relative_address" field is used when the picture is stored in the object (.aobj) file while memory is freed and reallocated to conserve space.

The other two structures are described in detail in the preceding sections.

As an example, consider the following recursively nested **SPACE** statement:

```
(00009): NR = 2;
(00010): SPACE = (NP,NFP,(UR:1..NR,UX:ARRAY[1..NR] OF BOOLEAN));
```

Figure 5 illustrates how the above SPACE statement will be stored in memory.

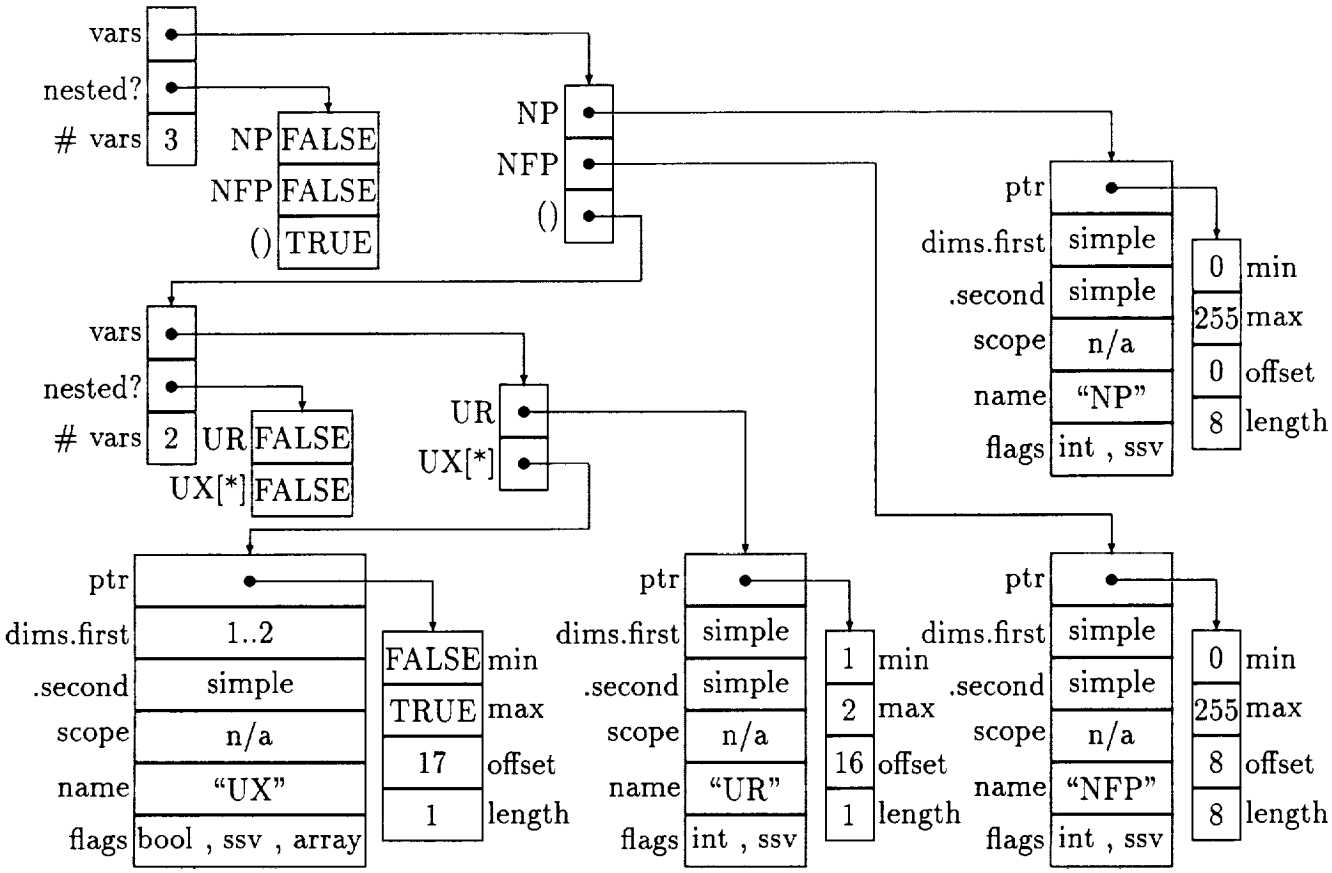



Figure 5: SPACE statement laid out in memory

For another example of a SPACE statement, consider:

```
(00009): NPMAX = 5;
(00010): SPACE = (NP:NPMAX..NPMAX,NFP:0..NPMAX,Q:BOOLEAN,
                  ELE:ARRAY[21..30] OF 20..23,NELE:0..10)
```

Figure 6 illustrates how the above SPACE statement will be stored in memory.

Notice that in Figure 6 the bit length for NP was one even though the value for NP is 5. Although the number 5 cannot be stored in one bit, the range 5..5 is of length one and the number 1 can be stored in one bit. Also note that ELE[ ] is said to be of length 2 because each element of ELE[ ] is of length 2. The array actually occupies 20 bits, two bits for each

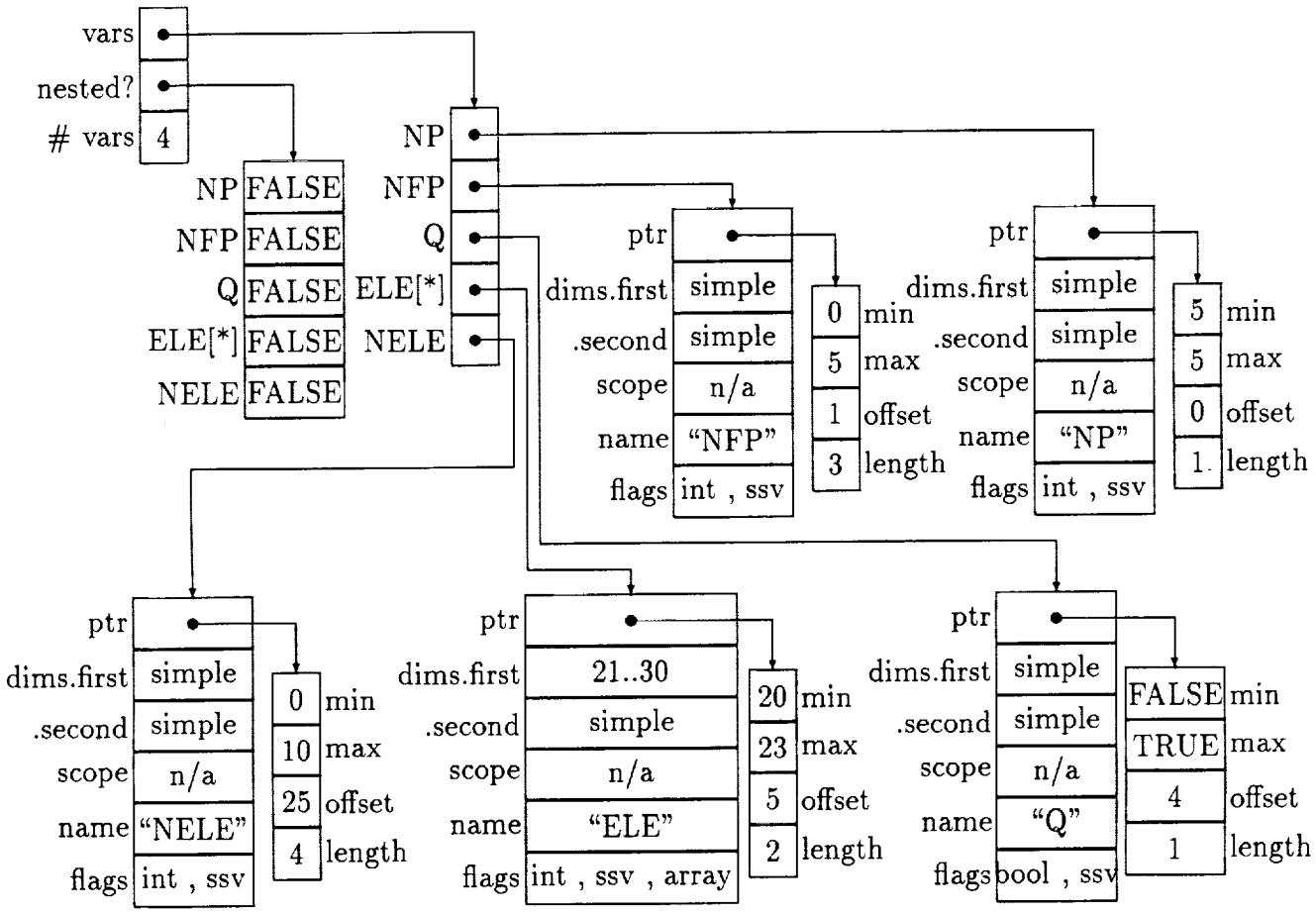



Figure 6: SPACE statement laid out in memory

of the ten elements in the array. If the offset of ELE[ ], which is 5, is compared to the offset of NELE, which is 25, the difference is 20, which is correct.

## 4.7 The START statement data structure

The START statement is parsed and stored in a data structure of the following type:

```
typedef struct t__space_expression
   {
       expression_type *exprs;
       operand_type *vars;
       short n_vars;
   } space_expression_type;
```

The "exprs" field is a pointer to an array of expressions. The array contains one expression for each scalar variable in the space, two for each singly subscripted array variable, and three for each doubly subscripted array variable. For arrays, the subscripts always precede the assigned value.

The "vars" field is a pointer to an array of pointers. Each pointer in the array of pointers points to an identifier in the identifier table. Each of the "vars" corresponds to one scalar l-value. There will be one var for each scalar state-space variable in the space and one for each element of each array in the space.

The "n_vars" field is the count of the number of scalar l-values in the START expression. Since the empty-field is not allowed in a START statement, this number should always equal the number of positions at its level in the nested state space.

As an example, consider the same recursively nested SPACE statement and corresponding START statement:

```
(00009): NR = 2;
(00010): SPACE = (NP,NFP,(UR:1..NR,UX:ARRAY[1..NR] OF BOOLEAN));
(00011): START = (1+2*NR,0,(2,NR OF NR<3));
```

The layout of the above START statement is illustrated in Figure 7.

Notice that, although $n_vars = 5$, there are seven expressions in the array pointed to by *exprs*. This is because two of the variables in the array pointed to by *vars* are for UX, which is an array variable. The array variable name UX is repeated in the *vars* list as many times as there are elements in the array.

Two expressions are stored in the array when an array variable is encountered in a positional state node. The first of the two expressions is always the subscript and the second is always the value to be stored in the state space.

After the START statement is parsed, an unconditional TRANTO is generated and stored for transition to the initial state.

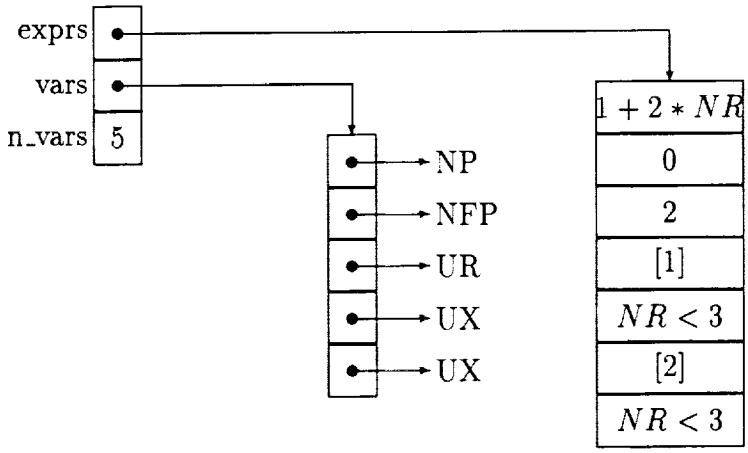



Figure 7: START statement laid out in memory

## 4.8 The current state bit string

When transitions between states occur, one bit string each is kept for both the source and destination states. Bit string lengths are rounded up so that each bit string will start on a byte boundary. Bit strings are packed for optimum use of memory. Register variables are used when packing and unpacking the bit strings to achieve maximum speed yet retain portability of the code. If the `ONEDEATH` option is `OFF`, then an extra bit is added before rounding to indicate a death state.

Bit strings are packed into an array pointed to by a pointer of the following type:

```
typedef unsigned char *state_bitstring_type;
```

Consider the following `SPACE` statement:

```
(00022): SPACE = (NP:0..6,NWP:3..6,Q:BOOLEAN,ELE:ARRAY[1..5] OF 10..25)
```

Two examples using the `SPACE` statement above are given in Figures 8 and 9. Both of these examples have an "unused" portion, which is necessary to align the next state on a byte boundary. If death states are not aggregated (lumped), then the first of these unused bits is used to flag the death states. In models where a state-space node exactly fits with no unused space, an extra byte is required for the flag bit when death states are not lumped. The default is to lump death states (`ONEDEATH ON`) according to the `DEATHIF` sequence in the input file.

Because the bound difference cannot exceed 32767 (fifteen bits) and because a byte is eight bits, a single state-space element cannot span more than three bytes. Because of this as-

$(6, 5, TRUE, 10, 12, 15, 24, 25)$

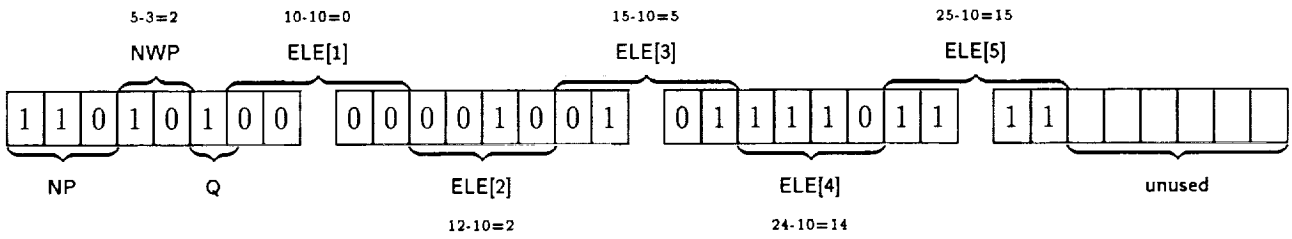



Figure 8: Example illustrating packing of a state-space node

$(6, 6, FALSE, 16, 25, 12, 25, 11)$

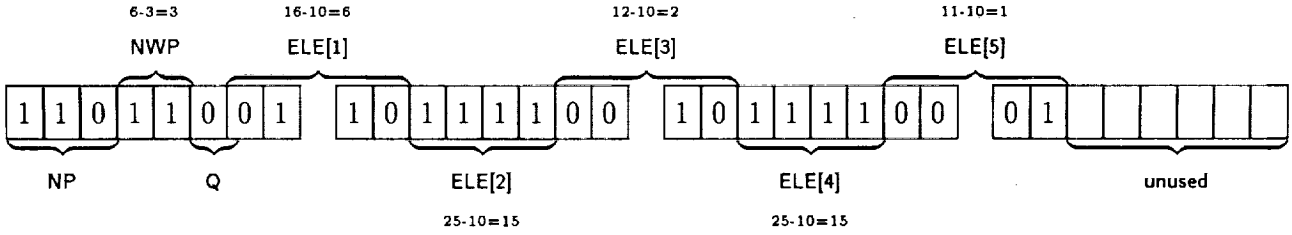



Figure 9: Another example of packing of a state-space node

sumption, the current code to pack and unpack the state bits would have to be modified to work correctly on 6-bit architectures such as CDC.

## 4.9 Macro definitions and data structures

There are two kinds of macro definitions in the ASSIST language. The first is the IMPLICIT variable definition. The second is the FUNCTION definition. The data structures for both of these are identical. Their only difference is that, after parsing the IMPLICIT variable definition, the formal parameter list is thrown away and the index list is retained. The formal parameter list is retained on the FUNCTION definition.

The semantics for macro definitions disallows reference to variables in the body of the definition except via the parameter list. Named constants, literal values, reserved words, and symbols may all be referenced in the body token list without having to be listed in the parameter list.

An IMPLICIT variable definition's parameter list may contain only state-space variables. Its index list may contain only dummy identifiers that are not space variables or named constants.

A FUNCTION definition's parameter list may contain only dummy identifiers that are not space variables or named constants.

The macro expansion stack is used for both IMPLICIT and FUNCTION definitions and consists of a stack of elements of the following data type:

```
typedef struct t__macro_expansion_info
   {
       token_info_type *passed_token_list;
       unsigned short *passed_token_offset;
       unsigned short *passed_token_counts;
       short now_passed_ix;
       short now_passed_count;
       short passed_parameter_count;
       short now_body_ix;
       short ovf_body_ix;
       short pos;
       short linnum;
   } macro_expansion_info_type;
```

where the data structure for a dimension pair is of the following type:

```
typedef struct t__dim_pair
  {
    Subscript lower;/* lower dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
    Subscript upper;/* upper dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
  } dim_pair_type;
```

and where the data structure for a pointer union is of the following type:

```
typedef union t__pointer_union
   {
     relative_address_type relative_address;
     short parameter_count;
     void *vvv;                /* included for completeness */
     Boolean *bbb;             /* used when BOOL_TYPE */
     state_offset_type *sss;   /* used when SSVAR_TYPE */
     char *ccc;                /* used when CHAR_TYPE */
     int_type *iii;            /* used when INT_TYPE */
     real_type *rrr;           /* used when REAL_TYPE */
   } pointer_union_type;
```

The first two sub-sections in this section deal with the definitions of the two kinds of macros. The third and last sub-section deals with how macros are expanded when they are invoked.

### 4.9.1 The IMPLICIT statement data structure

The IMPLICIT definition statement is parsed and stored in data structures of the following types:

```
typedef struct t__identifier_info
  {
     pointer_union_type ptr;    /* address in memory / function-parm-count */
     union
        {
           struct qqbothidinfqq
              {
                 dim_pair_type first;
                 dim_pair_type second;
              } dims;
           dim_pair_type body;
        } index;
     short scope_level;         /* scope level (negative iff. inactive) */
     char name[IDENT_MAXNCH_P]; /* identifier to search for */
     type_flagword_type flags;  /* type information */
  } identifier_info_type;
```

where the data structure for a dimension pair is of the following type:

```
typedef struct t__dim_pair
  {
    Subscript lower;/* lower dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
    Subscript upper;/* upper dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
  } dim_pair_type;
```

and where the data structure for a pointer union is of the following type:

```
typedef union t__pointer_union
   {
     relative_address_type relative_address;
     short parameter_count;
     void *vvv;                /* included for completeness */
     Boolean *bbb;             /* used when BOOL_TYPE */
```

```
    state_offset_type *sss;  /* used when SSVAR_TYPE */
    char *ccc;               /* used when CHAR_TYPE */
    int_type *iii;           /* used when INT_TYPE */
    real_type *rrr;          /* used when REAL_TYPE */
  } pointer_union_type;

typedef struct t__token_info

    identifier_info_type *id_info_ptr;
    short linnum;
    short pos;
    token tok;
    rwtype rw;
    char id[IDENT_MAXNCH_P];
  token_info_type;
```

and the body is stored in the array pointed to by:

```
extern token_info_type *function_body_storage;
```

which is allocated dynamically before the parse phase begins and freed again before the rule generation phase begins.

### 4.9.2 The FUNCTION statement data structure

The FUNCTION definition statement is parsed and stored in data structures of the following types:

```
typedef struct t__identifier_info
  {
    pointer_union_type ptr;    /* address in memory / function-parm-count */
    union
      {
        struct qqbothidinfqq
          {
            dim_pair_type first;
            dim_pair_type second;
          } dims;
        dim_pair_type body;
      } index;
    short scope_level;         /* scope level (negative iff. inactive) */
    char name[IDENT_MAXNCH_P]; /* identifier to search for */
    type_flagword_type flags;  /* type information */
  } identifier_info_type;
```

where the data structure for a dimension pair is of the following type:

```
typedef struct t__dim_pair
  {
    Subscript lower;/* lower dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
    Subscript upper;/* upper dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
  } dim_pair_type;
```

and where the data structure for a pointer union is of the following type:

```
typedef union t__pointer_union
   {
     relative_address_type relative_address;
     short parameter_count;
     void *vvv;                  /* included for completeness */
     Boolean *bbb;               /* used when BOOL_TYPE */
     state_offset_type *sss;     /* used when SSVAR_TYPE */
     char *ccc;                  /* used when CHAR_TYPE */
     int_type *iii;              /* used when INT_TYPE */
     real_type *rrr;             /* used when REAL_TYPE */
   } pointer_union_type;

typedef struct t__token_info

      identifier_info_type *id_info_ptr;
      short linnum;
      short pos;
      token tok;
      rwtype rw;
      char id[IDENT_MAXNCH_P];
    token_info_type;
```

and the body is stored in the array pointed to by:

```
extern token_info_type *function_body_storage;
```

which is allocated dynamically before the parse phase begins and freed again before the rule generation phase begins.

As an example, consider the following FUNCTION definition statement:

```
(00012): MAXVAL = 10;
(00013): FUNCTION FOO(I,X) = X**(MAXVAL-I);
```

Figure 10 illustrates how the above FUNCTION definition statement will be stored in memory.

Note that the upper index is one greater than the last index for the parameter list so that a loop on the parameters can continue while the index is strictly less than this value.

### 4.9.3 The macro expansion stack data structure

When macros (IMPLICIT variables and FUNCTIONs) are expanded, the body of the expansion is pushed onto the "macro expansion stack". When parsing advances to the next token, the macro expansion stack is first checked before reading from the input stream. If the stack is non-empty, then the next token in the body on the top of the stack is taken. If the body list on top of the stack has already been exhausted, then the stack is popped. When the stack is empty, the next token is read from the input stream.

The macro expansion stack consists of a stack of elements of the following data type:

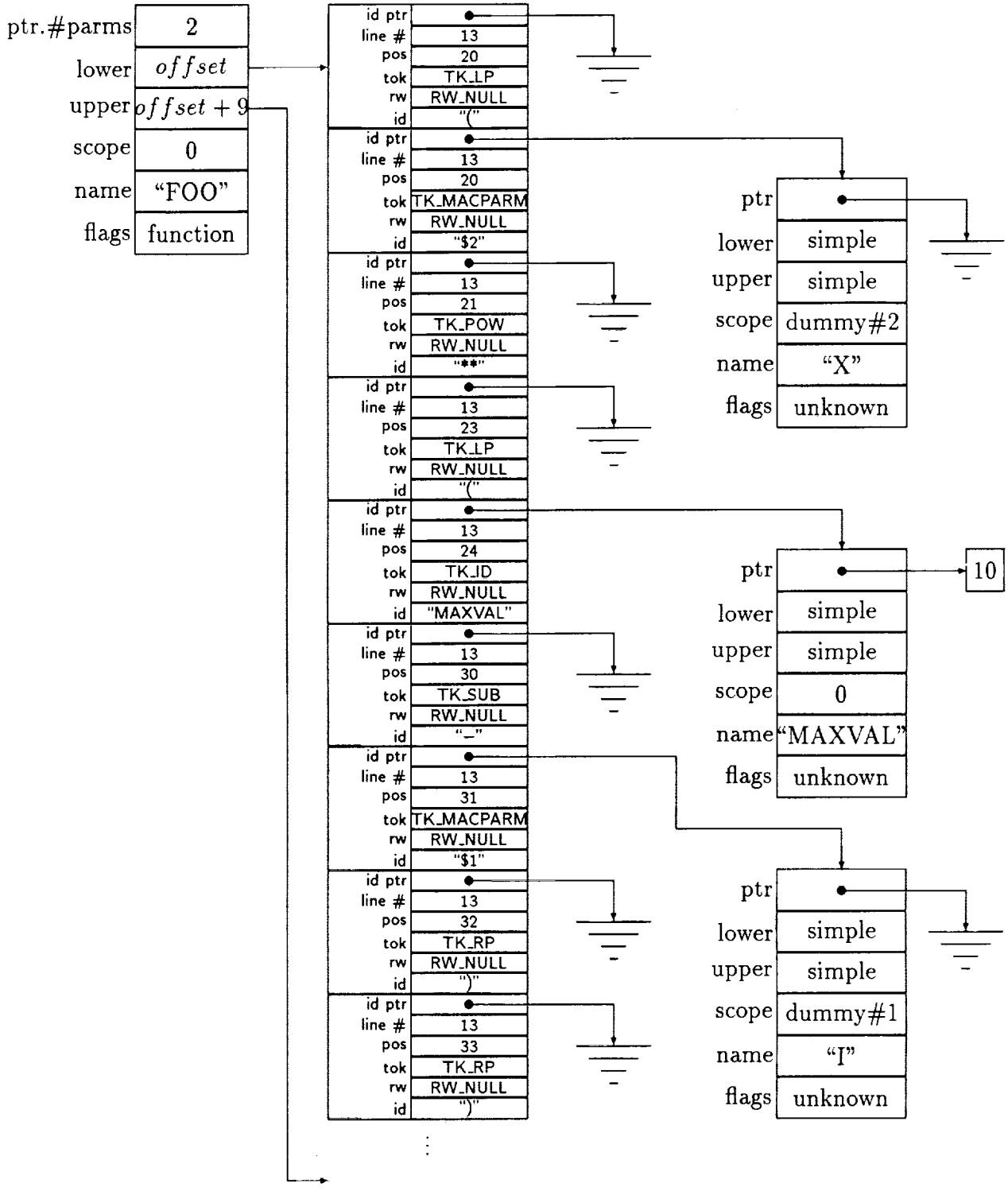



Figure 10: Layout of FUNCTION definition in memory

```
typedef struct t__macro_expansion_info
   {
       token_info_type *passed_token_list;
       unsigned short *passed_token_offset;
       unsigned short *passed_token_counts;
       short now_passed_ix;
       short now_passed_count;
       short passed_parameter_count;
       short now_body_ix;
       short ovf_body_ix;
       short pos;
       short linnum;
   } macro_expansion_info_type;
```

where the data structure for a dimension pair is of the following type:

```
typedef struct t__dim_pair
  {
    Subscript lower;/* lower dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
    Subscript upper;/* upper dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
  } dim_pair_type;
```

and where the data structure for a pointer union is of the following type:

```
typedef union t__pointer_union
   {
     relative_address_type relative_address;
     short parameter_count;
     void *vvv;                /* included for completeness */
     Boolean *bbb;             /* used when BOOL_TYPE */
     state_offset_type *sss;   /* used when SSVAR_TYPE */
     char *ccc;                /* used when CHAR_TYPE */
     int_type *iii;            /* used when INT_TYPE */
     real_type *rrr;           /* used when REAL_TYPE */
   } pointer_union_type;
```

Each macro expansion information record contains an array of lists of calling expression tokens as well as a pointer to the list of tokens making up the body of the macro. For example, consider:

```
(00070): FUNCTION F(X) = X * (X-2.0*X);
(00071): FUNCTION G(A,B,C) = A * (B - C);
         ...
(00099): ...  + G(F(2.0+QQQ),F(2.0-QQQ),F((Q1+Q2)/2.0)) ...
```

In the previous invocation of function G() there are nested invocations of function F(). During the parsing of $F(2.0 + QQQ)$, a brief description of the contents of the stack is diagrammed in Figure 11. A more detailed description using a simpler example will appear later.

When a parameter reference, such as \$1, is encountered, the "now_passed_ix" is changed from negative one less the offset to the index of the parameter (0 for \$1, 1 for \$2, 2 for \$3, etc.). When all tokens for the parameter have been exhausted, then the index is changed

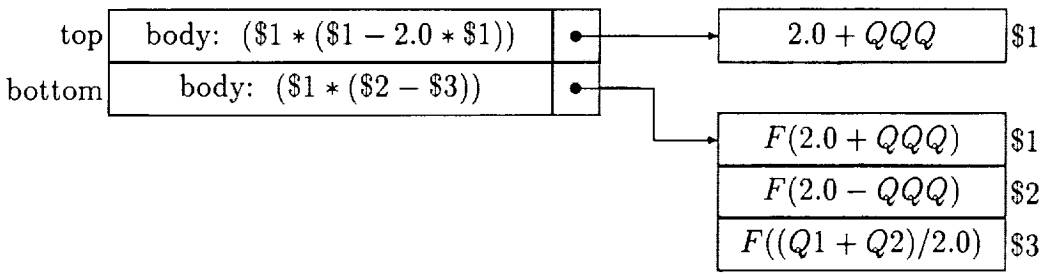



Figure 11: Overview of expansion stack during nested function invocation

back to negative one less the offset and the next token is taken from the body once again. When all tokens for the body have been exhausted, then the stack is popped.

The arithmetic "negative one less the offset" is used because the number zero is a valid offset and negative zero is the same as positive zero. The extra offset of negative one is therefore necessary so that the set of negatives is disjoint from the set of positives.

Parentheses are inserted when the count of the number of tokens making up a calling parameter is greater than one. For example, the following translations will be performed:

$$F(MU) \longrightarrow MU * (MU - 2.0)$$
$$F(A + B) \longrightarrow (A + B) * ((A + B) - 2.0)$$

In the above examples, no parentheses were added in the first example because the calling parameter "$MU$" is only one token long. Parentheses were added in the second example since "$A + B$" is three tokens long (more than one).

The following data types are referenced by the elements on the stack:

```
typedef struct t__token_info

      identifier_info_type *id_info_ptr;
      short linnum;
      short pos;
      token tok;
      rwtype rw;
      char id[IDENT_MAXNCH_P];
   token_info_type;

typedef struct t__identifier_info
  {
     pointer_union_type ptr;     /* address in memory / function-parm-count */
     union
        {
```

```
            struct qqbothidinfqq
               {
                  dim_pair_type first;
                  dim_pair_type second;
               } dims;
            dim_pair_type body;
         } index;
      short scope_level;          /* scope level (negative iff. inactive) */
      char name[IDENT_MAXNCH_P]; /* identifier to search for */
      type_flagword_type flags;  /* type information */
   } identifier_info_type;
```

where the data structure for a dimension pair is of the following type:

```
typedef struct t__dim_pair
  {
    Subscript lower;/* lower dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
    Subscript upper;/* upper dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
  } dim_pair_type;
```

and where the data structure for a pointer union is of the following type:

```
typedef union t__pointer_union
   {
     relative_address_type relative_address;
     short parameter_count;
     void *vvv;                /* included for completeness */
     Boolean *bbb;             /* used when BOOL_TYPE */
     state_offset_type *sss;   /* used when SSVAR_TYPE */
     char *ccc;                /* used when CHAR_TYPE */
     int_type *iii;            /* used when INT_TYPE */
     real_type *rrr;           /* used when REAL_TYPE */
   } pointer_union_type;
```

As an example, consider the following FUNCTION definition statement:

```
(00012): MAXVAL = 10;
(00013): FUNCTION FOO(I,X) = X**(MAXVAL-I);
```

and the following reference to this function:

```
(00016): IF (...) TRANTO ...
(00017):    BY FOO(12-2*IX,OMEGA/LAMBDA) ...
```

The illustration in Figure 12 details how the above FUNCTION definition statement will be pushed onto the macro expansion stack.

Note that all of the identifier table pointers are null. This is because passed parameters are just sequences of tokens. No identifier information is needed when a token is pulled off of the macro expansion stack. Identifier information is always looked up after a token is retrieved regardless of whether it is retrieved from the macro expansion stack or from the input file.

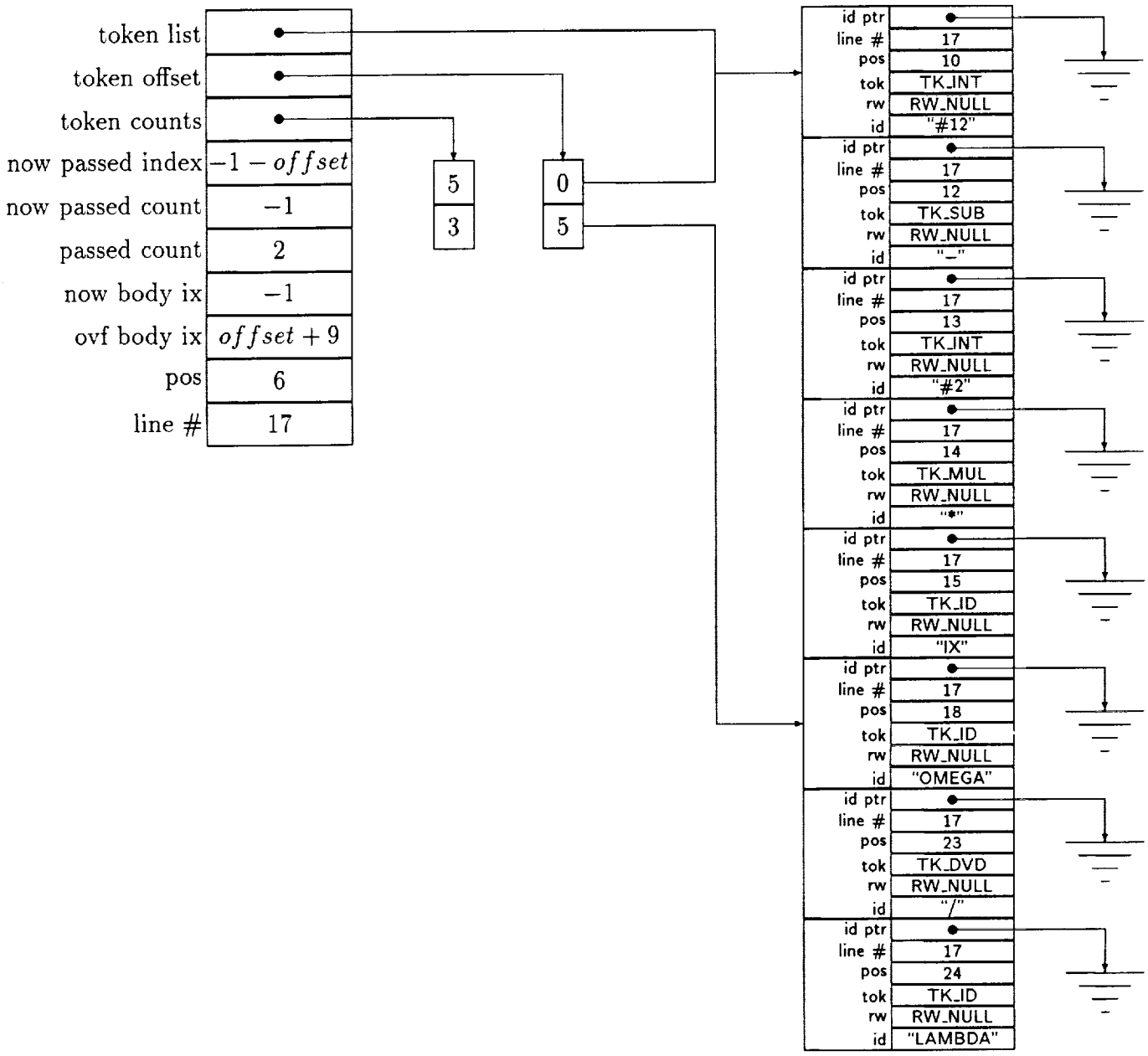



Figure 12: Detail of expansion stack during function invocation

## 4.10 The VARIABLE statement data structure

The VARIABLE statements are parsed and stored in data stuctures of the following types:

```
typedef struct t__calc_assign
   {
       identifier_info_type *idinfo;  /* <ident> := */
       expression_type *expr;         /*            <expr> */
   } calc_assign_type;

typedef struct t__expression
   {
       operation_type *postfix_ops;
       operation_type *infix_ops;
       operand_type *operands;
       short n_postfix_ops;
       short n_infix_ops;
       short n_operands;
       short source_code_line_number;
       Boolean in_error;
       type_flagword_type rtntype;
   } expression_type;

typedef struct t__identifier_info
  {
     pointer_union_type ptr;    /* address in memory / function-parm-count */
     union
        {
           struct qqbothidinfqq
              {
                 dim_pair_type first;
                 dim_pair_type second;
              } dims;
           dim_pair_type body;
        } index;
     short scope_level;         /* scope level (negative iff. inactive) */
     char name[IDENT_MAXNCH_P]; /* identifier to search for */
     type_flagword_type flags;  /* type information */
  } identifier_info_type;
```

where the data structure for a dimension pair is of the following type:

```
typedef struct t__dim_pair
  {
    Subscript lower;/* lower dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
    Subscript upper;/* upper dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
  } dim_pair_type;
```

and where the data structure for a pointer union is of the following type:

```
typedef union t__pointer_union
   {
     relative_address_type relative_address;
     short parameter_count;
     void *vvv;                 /* included for completeness */
```

```
    Boolean *bbb;              /* used when BOOL_TYPE */
    state_offset_type *sss;    /* used when SSVAR_TYPE */
    char *ccc;                 /* used when CHAR_TYPE */
    int_type *iii;             /* used when INT_TYPE */
    real_type *rrr;            /* used when REAL_TYPE */
  } pointer_union_type;
```

As an example, consider:

```
(00010): VARIABLE NWP[NFP] = NP-NFP;
```

The layout of the above `VARIABLE` is illustrated in Figure 13.

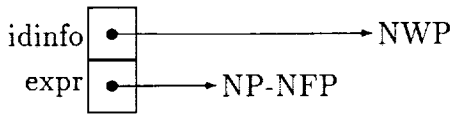



Figure 13: VARIABLE statement laid out in memory

## 4.11 The cross-reference-map entry data structure

When a cross-reference map is requested with the **−xref** command line option, a file is created with entries of the following type:

```
typedef struct t__cross_reference_entry
  {
    unsigned short linnum;
    unsigned short pos;
    char name[XREF_IDENT_MAXNCH_P];
    char refcode;
  } cross_reference_entry_type;
```

The "linnum" field holds the ASSIST input file line number as listed in the log file.

The "pos" field holds the character column position on the line where the name begins.

The "name" field holds the name of the item being cross referenced. Usually the name is just the identifier name. Cross reference listings also detail `ELSE's` and `ENDIF's` with their corresponding `IF's` as well as `ENDFOR's` with their corresponding `FOR's`.

The "refcode" is a character that indicates the type of reference:

- 'D' stands for a declaration.
- 'S' stands for "set" and indicates where the "name" takes on a value.
- 'U' stands for "use" and indicates where the value of "name" is used to compute something else.

# 5 Data Structures Used to Generate a Model File

This section describes the data structures used to generate the reliability model.

## 5.1 Introduction to model generation data structures

Four copies of the rule section code are stored in memory for fast and efficient generation of a model. The four copies are for:

- `ASSERT` assertions
- `DEATHIF` failures
- `PRUNEIF` checks
- `TRANTO` transitions

These four copies of the rule section are preceded by a preamble section of code, which contains the `START` state transition and pointers to the four copies of the rule code listed above.

Each of the four copies has code for its own type in addition to code for other types not in a copy of their own. For example, the assertion copy of the code contains ASSERT, IF, and FOR statements as well as some internal statements (which are not part of the ASSIST input language). such as GOTO, GOSUB and RETURN. Operation codes are detailed in Chapter 3 on page 8.

The body of a block `IF` and the body of a `FOR` are implemented in the pseudo-code language as a subroutine. The `THEN`, `ELSE`, and `FOR` keywords imply a GOSUB instruction. The `ELSE`, `ENDIF`, and `ENDFOR` imply the RETURN instruction corresponding to the respective keywords implying the GOSUB instruction.

Code is stored in data structures of the following types:

```
typedef struct t__instruction_pointer_union
   {
       void *vvv;                        /* to cast to block_if_type, etc. */
       relative_address_type reladdr;    /* relative address of code */
   } instruction_pointer_union_type;

typedef struct t__instruction
   {
       instruction_pointer_union_type ptr;
       opcode_type opcode;       /* instruction operation code */
   } instruction_type;           /*    for_loop_type, assert_type, ... */

typedef struct t__block_if
   {
       expression_type *then_test;    /* boolean expression for THEN */
```

```
        instruction_type *then_clause; /* code for THEN clause */
        instruction_type *else_clause; /* code for ELSE clause */
    } block_if_type;

typedef struct t__set_range
    {
        expression_type *lower_bound;
        expression_type *upper_bound;
    } set_range_type;

typedef struct t__for_loop
    {
        identifier_info_type *ident;  /* index variable */
        set_range_type *set_ranges;   /* pointer to array of IN ranges */
        short set_range_count;        /* count of number of IN ranges */
        instruction_type *body;       /* pointer to BODY of loop */
    } for_loop_type;

typedef struct t__state_space_picture
    {
        vars_union_type *varu;
        Boolean *is_nested;
        short nvaru;
    } state_space_picture_type;

typedef union t__vars_union
    {
        identifier_info_type *id_info;
        state_space_picture_type *nested_space_picture;
        relative_address_type relative_address;
    } vars_union_type;

typedef union t__node_union
    {
        state_space_picture_type pix;      /* 10 bytes  (max(10,10)=10 bytes) */
    } node_union_type;

typedef struct t__space_expression
    {
        expression_type *exprs;
        operand_type *vars;
        short n_vars;
    } space_expression_type;

typedef struct t__tranto_clause
    {
        space_expression_type sex;   /* list of space transition expressions */
        expression_type *rate_exprs; /* ptr to array of rate expressions */
        short n_rate_exprs;          /* count of rate expressions */
        short source_code_line_number;
    } tranto_clause_type;

typedef struct t__booltest
    {
        expression_type *expr;         /* boolean expr to ASSERT,DEATHIF,etc. */
        short source_code_line_number; /* line number in listing file */
        short lumping_sequence;        /* sequence index (0..n-1) in source */
    } booltest_type;                   /* e.g., first DEATHIF, second DEATHIF */
```

```
typedef booltest_type assert_type;

typedef booltest_type deathif_type;

typedef booltest_type pruneif_type;
```

Note that the pointer "void *vvv" in the "instruction_pointer_union_type" data type is cast to a pointer of the appropriate type as illustrated for "instruction_type inst;" in the following examples:

```
(block_if_type *) inst.vvv
(for_loop_type *) inst.vvv
(tranto_clause_type *) inst.vvv
(assert_type *) inst.vvv
(deathif_type *) inst.vvv
(pruneif_type *) inst.vvv
```

Examples of instructions are given in the following sections.

## 5.2 The ASSERT statement data structure

The ASSERT statement is parsed and stored in data structures of the following types:

```
typedef struct t__instruction_pointer_union
   {
       void *vvv;                          /* to cast to block_if_type, etc. */
       relative_address_type reladdr;      /* relative address of code */
   } instruction_pointer_union_type;

typedef struct t__instruction
   {
       instruction_pointer_union_type ptr;
       opcode_type opcode;          /* instruction operation code */
   } instruction_type;              /*    for_loop_type, assert_type, ... */

typedef struct t__booltest
   {
       expression_type *expr;           /* boolean expr to ASSERT,DEATHIF,etc. */
       short source_code_line_number; /* line number in listing file */
       short lumping_sequence;          /* sequence index (0..n-1) in source */
   } booltest_type;                     /* e.g., first DEATHIF, second DEATHIF */

typedef booltest_type assert_type;
```

The "expr" field is a pointer to the expression to test for conformance.

The "source_code_line_number" indicates the line number in the source code (*".ast"*) file as listed in the log (*".alog"*) file where the statement began. It is used to print intelligent error messages during both parsing and model generation phases.

The "sequence_index" gives the ASSERT sequence number. This number is zero for the first ASSERT that is parsed, one for the second ASSERT that is parsed, etc. The sequence number is used in order to lump death and prune states. The sequence number is defined for ASSERT statements but is currently not referenced since ASSERT statements are not lumped. It is independent of the number of DEATHIF and PRUNEIF statements encountered since they have their own sequence index counters.

As an example, consider the following log file excerpt showing only the ASSERT statements:

```
(00099): ASSERT ...
         :
(00112): ASSERT ...
(00113): ASSERT ...
         :
(00125): ASSERT ...
(00126): ASSERT ...
         :
(00139): ASSERT ...
         :
(00147): ASSERT NP>NFP;
```

Figure 14 pictures the memory layout of the ASSERT from line 147. Note that the sequence index is six because sequences begin with the number zero.

For a more complete illustration of how an expression is laid out in memory, see Figure 4 on page 23.

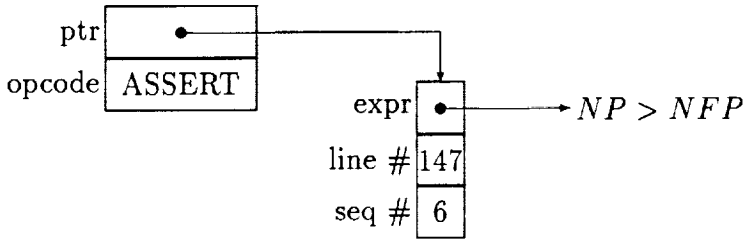



Figure 14: Sample ASSERT laid out in memory

## 5.3 The DEATHIF statement data structure

The DEATHIF statement is parsed and stored in data structures of the following types:

```
typedef struct t__instruction_pointer_union
   {
```

```
        void *vvv;                          /* to cast to block_if_type, etc. */
        relative_address_type reladdr;      /* relative address of code */
    } instruction_pointer_union_type;

typedef struct t__instruction
    {
        instruction_pointer_union_type ptr;
        opcode_type opcode;         /* instruction operation code */
    } instruction_type;             /*    for_loop_type, assert_type, ... */

typedef struct t__booltest
    {
        expression_type *expr;              /* boolean expr to ASSERT,DEATHIF,etc. */
        short source_code_line_number;      /* line number in listing file */
        short lumping_sequence;             /* sequence index (0..n-1) in source */
    } booltest_type;                        /* e.g., first DEATHIF, second DEATHIF */

typedef booltest_type deathif_type;
```

The "expr" field is a pointer to the expression to test for conformance.

The "source_code_line_number" indicates the line number in the source code (*".ast"*) file as listed in the log (*".alog"*) file where the statement began. It is used to print intelligent error messages during both parsing and model generation phases.

The "sequence_index" gives the DEATHIF sequence number. This number is zero for the first DEATHIF that is parsed, one for the second DEATHIF parsed, etc. The sequence number is used in order to lump death and prune states. It is independent of the number of ASSERT and PRUNEIF statements encountered since they have their own sequence index counters.

As an example, consider the following DEATHIF statement preceded by an IMPLICIT definition that is referenced in the DEATHIF:

```
(00146): IMPLICIT NWP(NP,NFP) = NP-NFP;
(00147): DEATHIF NFP>NWP;
```

Figure 15 pictures the memory layout of the DEATHIF from line 147. Note that the sequence index is zero because sequences begin with the number zero and there are no DEATHIF statements preceding it. Notice also that the IMPLICIT variable macro expansion was made before the expression was parsed and stored.

For a more complete illustration of how an expression is laid out in memory, see Figure 4 on page 23.

## 5.4 The PRUNEIF statement data structure

The PRUNEIF statement is parsed and stored in data structures of the following types:

```
typedef struct t__instruction_pointer_union
    {
```

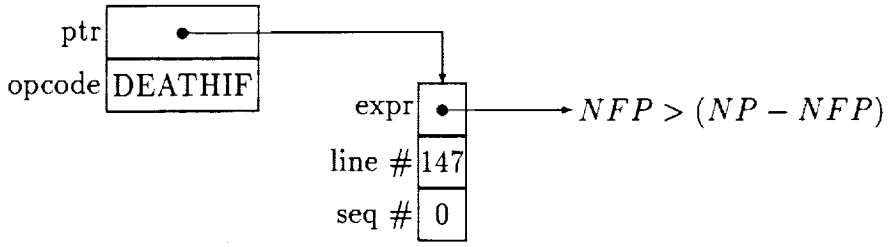



Figure 15: Sample DEATHIF laid out in memory

```
        void *vvv;                          /* to cast to block_if_type, etc. */
        relative_address_type reladdr;      /* relative address of code */
    } instruction_pointer_union_type;

typedef struct t__instruction
    {
        instruction_pointer_union_type ptr;
        opcode_type opcode;         /* instruction operation code */
    } instruction_type;             /*    for_loop_type, assert_type, ... */

typedef struct t__booltest
    {
        expression_type *expr;              /* boolean expr to ASSERT,DEATHIF,etc. */
        short source_code_line_number;      /* line number in listing file */
        short lumping_sequence;             /* sequence index (0..n-1) in source */
    } booltest_type;                        /* e.g., first DEATHIF, second DEATHIF */

typedef booltest_type pruneif_type;
```

The "expr" field is a pointer to the expression to test for conformance.

The "source_code_line_number" indicates the line number in the source code (*".ast"*) file as listed in the log (*".alog"*) file where the statement began. It is used to print intelligent error messages during both parsing and model generation phases.

The "sequence_index" gives the PRUNEIF sequence number. This number is zero for the first PRUNEIF that is parsed, one for the second PRUNEIF parsed, etc. The sequence number is used in order to lump death and prune states. It is independent of the number of ASSERT and DEATHIF statements encountered since they have their own sequence index counters.

As an example, consider the following PRUNEIF statements:

```
(00101): PRUNEIF  ...
         :
(00122): PRUNEIF  ...
         :
(00177): PRUNEIF NFP>3;
```

Figure 16 illustrates the memory layout of the PRUNEIF on line 177. Note that the sequence index is two because sequences begin with the number zero.

For a more complete illustration of how an expression is laid out in memory, see Figure 4 on page 23.

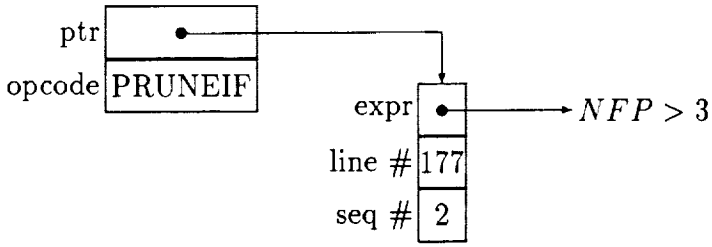



Figure 16: Sample PRUNEIF laid out in memory

## 5.5 The TRANTO statement data structure

The TRANTO statement clause is parsed and stored in a data structures of the following types:

```
typedef struct t__tranto_clause
   {
       space_expression_type sex;   /* list of space transition expressions */
       expression_type *rate_exprs; /* ptr to array of rate expressions */
       short n_rate_exprs;          /* count of rate expressions */
       short source_code_line_number;
   } tranto_clause_type;

typedef struct t__space_expression
   {
       expression_type *exprs;
       operand_type *vars;
       short n_vars;
   } space_expression_type;
```

There are two formats for the TRANTO clause destination. The first format is a list of assignment statements. The second format is a space expression.

An example of the first (list) format follows:

```
(00009): NR = 2;
(00010): SPACE = (NP,NFP,(UR:1..NR,UX:ARRAY[1..NR] OF BOOLEAN));
         :
(00017):  ...  TRANTO NFP++,UX[NR+1-III]=TRUE BY FAST III*DELTA;
```

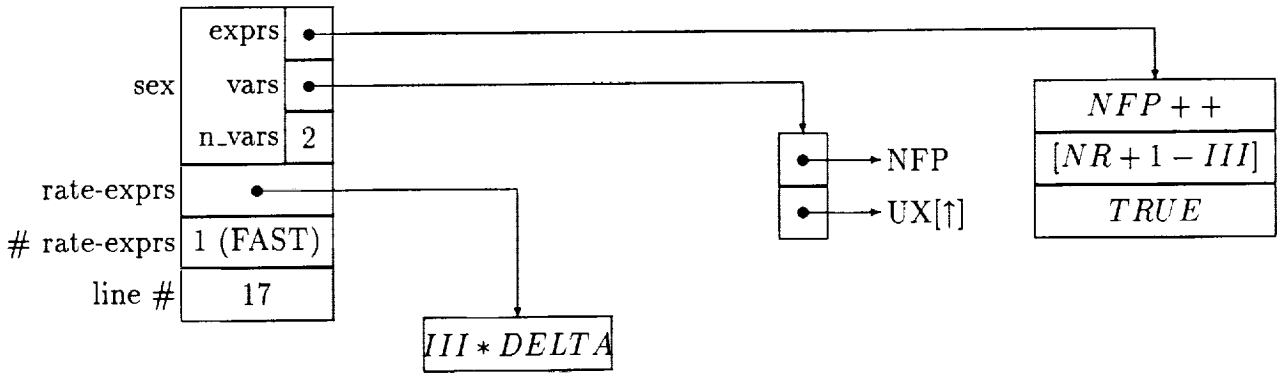



Figure 17: TRANTO clause (list format) laid out in memory

This list format example above is illustrated in Figure 17.

Notice that, although $n_vars = 2$, there are three expressions in the array pointed to by *sex.exprs*. This is because one of the variables in the array pointed to by *sex.vars* is for UX, which is an array variable. Two expressions are stored in the array when an array variable is encountered in a positional destination state description. The first of the two expressions is always the subscript and the second is always the value to be stored in the state space.

An example of the second (space expression) format follows:

```
(00009): NR = 2;
(00010): SPACE = (NP,NFP,(UR:1..NR,UX:ARRAY[1..NR] OF BOOLEAN));
          ⋮
(00017):  ...  TRANTO (,NFP+1,(,2 OF TRUE)) BY FAST III*DELTA;
```

Notice that, in a positional destination state description, multiple commas in a row indicate that the values for the state-space variable(s) occupying the position(s) in question are to remain unchanged after the state transition is made. This space expression format example above is illustrated in Figure 18.

## 5.6 The VARIABLE statement data structure

The VARIABLE statements are parsed and stored in data stuctures of the following types:

```
typedef struct t__calc_assign
   {
       identifier_info_type *idinfo;  /* <ident> := */
```

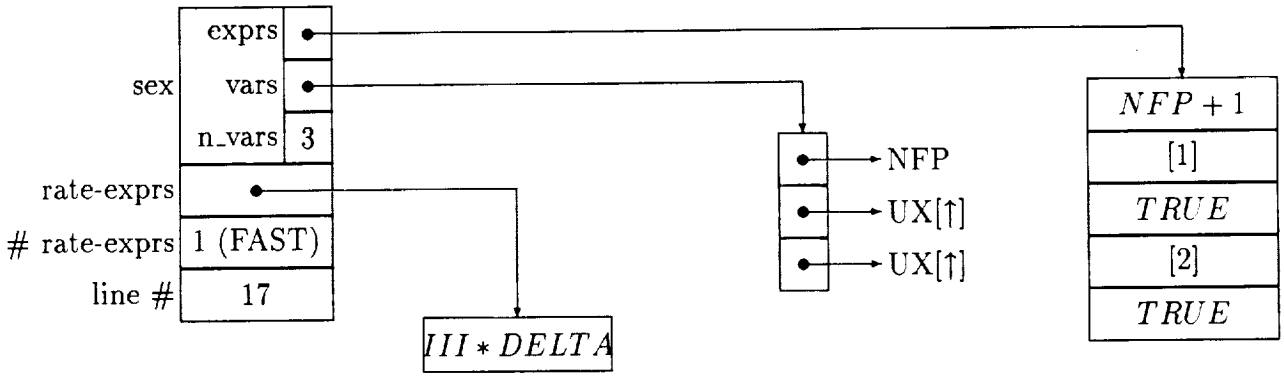



Figure 18: TRANTO clause (space expression format) laid out in memory

```
        expression_type *expr;          /*              <expr> */
    } calc_assign_type;

typedef struct t__expression
    {
        operation_type *postfix_ops;
        operation_type *infix_ops;
        operand_type *operands;
        short n_postfix_ops;
        short n_infix_ops;
        short n_operands;
        short source_code_line_number;
        Boolean in_error;
        type_flagword_type rtntype;
    } expression_type;

typedef struct t__identifier_info
    {
        pointer_union_type ptr;     /* address in memory / function-parm-count */
        union
            {
                struct qqbothidinfqq
                    {
                        dim_pair_type first;
                        dim_pair_type second;
                    } dims;
                dim_pair_type body;
            } index;
        short scope_level;          /* scope level (negative iff. inactive) */
        char name[IDENT_MAXNCH_P]; /* identifier to search for */
        type_flagword_type flags;  /* type information */
    } identifier_info_type;
```

where the data structure for a dimension pair is of the following type:

```
typedef struct t__dim_pair
    {
```

```
    Subscript lower;/* lower dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
    Subscript upper;/* upper dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
  } dim_pair_type;
```

and where the data structure for a pointer union is of the following type:

```
typedef union t__pointer_union
   {
     relative_address_type relative_address;
     short parameter_count;
     void *vvv;                /* included for completeness */
     Boolean *bbb;             /* used when BOOL_TYPE */
     state_offset_type *sss;   /* used when SSVAR_TYPE */
     char *ccc;                /* used when CHAR_TYPE */
     int_type *iii;            /* used when INT_TYPE */
     real_type *rrr;           /* used when REAL_TYPE */
   } pointer_union_type;
```

As an example, consider:

```
(00010): VARIABLE NWP[NFP] = NP-NFP;
```

The layout of the above **VARIABLE** is illustrated in Figure 19.

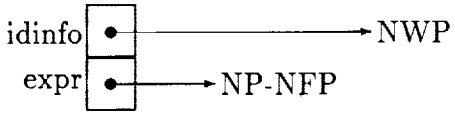



Figure 19: VARIABLE statement laid out in memory

## 5.7 The Block IF statement data structure

The block IF statement is parsed and stored in data structures of the following types:

```
typedef struct t__block_if
   {
       expression_type *then_test;    /* boolean expression for THEN */
       instruction_type *then_clause; /* code for THEN clause */
       instruction_type *else_clause; /* code for ELSE clause */
   } block_if_type;

typedef struct t__expression
   {
       operation_type *postfix_ops;
       operation_type *infix_ops;
       operand_type *operands;
       short n_postfix_ops;
       short n_infix_ops;
```

```
        short n_operands;
        short source_code_line_number;
        Boolean in_error;
        type_flagword_type rtntype;
    } expression_type;

typedef struct t__instruction_pointer_union
    {
        void *vvv;                          /* to cast to block_if_type, etc. */
        relative_address_type reladdr;      /* relative address of code */
    } instruction_pointer_union_type;

typedef struct t__instruction
    {
        instruction_pointer_union_type ptr;
        opcode_type opcode;         /* instruction operation code */
    } instruction_type;             /*    for_loop_type, assert_type, ... */
```

The "then_test" field is a pointer to a Boolean expression to be evaluated in order to decide whether to execute the THEN or ELSE clause code. If the expression pointed to evaluates to TRUE, then the THEN clause code is executed, otherwise the ELSE clause code is executed.

The "then_clause" field is a pointer to the beginning of the subroutine that contains the rule-section instructions of the THEN clause and pertaining to the current code section, i.e., a block IF in the DEATHIF section will have a THEN clause that points to a subroutine in the DEATHIF section.

The "else_clause" field is a pointer to the beginning of the subroutine which contains the rule-section instructions of the ELSE clause and pertaining to the current code section.

As an example, consider the following block if:

```
(0010):    IF B=6 THEN;
(0011):       TRANTO A=A-1 BY FOO1;
(0012):       TRANTO A=A-1,B=B-1 BY FOO2;
(0013):    ENDIF;
```

Figure 20 illustrates the memory layout of the block IF beginning on line 10. This memory layout is also detailed with an excerpt from the memory load map ( **—loadmap** option) in Figure 21. Note that the ENDIF matches the THEN and not the IF.

## 5.8 The FOR loop statement data structure

The FOR statement is parsed and stored in data structures of the following types:

```
typedef struct t__for_loop
    {
        identifier_info_type *ident;  /* index variable */
        set_range_type *set_ranges;   /* pointer to array of IN ranges */
        short set_range_count;        /* count of number of IN ranges */
        instruction_type *body;       /* pointer to BODY of loop */
    } for_loop_type;
```

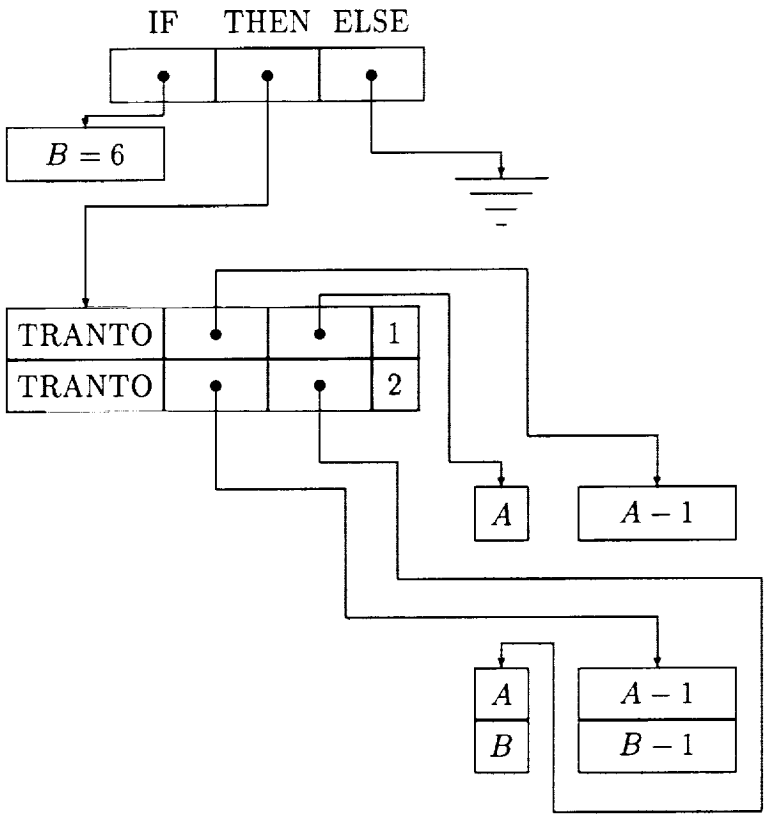



Figure 20: Sample block IF laid out in memory

```
00000211:   [post=(00000465,3),in=(00000468,3),op=(000003B7,2),
             line#=10,err=F,returns(0x0a,(<boolean>,<expr-var>))]
             B=6
00000227:   [post=(0000046B,3),in=(0000046E,3),op=(000003BF,2),
             line#=11,err=F,returns(0x0b,(<integer>,<expr-var>))]
            A-1
00000253:   [post=(00000473,3),in=(00000476,3),op=(000003CF,2),
             line#=12,err=F,returns(0x0b,(<integer>,<expr-var>))]
            A-1
00000269:   [post=(00000479,3),in=(0000047C,3),op=(000003D7,2),
             line#=12,err=F,returns(0x0b,(<integer>,<expr-var>))]
            B-1
0000023D:   [post=(00000471,1),in=(00000472,1),op=(000003CB,1),
             line#=11,err=F,returns(0x04,(<real>))]
            FOO1
0000027F:   [post=(0000047F,1),in=(00000480,1),op=(000003E7,1),
             line#=12,err=F,returns(0x04,(<real>))]
            FOO2

000003C7:   A<6>
000003DF:   A<6>
000003E3:   B<7>

000004FF:   (TRANTO (exprs=00000227,vars=000003C7,#vars=1)
                 BY 1@0000023D (line#11))
00000511:   (TRANTO (exprs=00000253,vars=000003DF,#vars=2)
                 BY 1@0000027F (line#12))

0000055F:   (IF 00000211 THEN GOSUB 00000619)

00000619:   TRANTO 000004FF
00000623:   TRANTO 00000511
0000062D:   RETURN
```

Figure 21: Sample memory map of corresponding block IF

```
typedef struct t__set_range
   {
       expression_type *lower_bound;
       expression_type *upper_bound;
   } set_range_type;

typedef struct t__expression
   {
       operation_type *postfix_ops;
       operation_type *infix_ops;
       operand_type *operands;
       short n_postfix_ops;
       short n_infix_ops;
       short n_operands;
       short source_code_line_number;
       Boolean in_error;
       type_flagword_type rtntype;
   } expression_type;

typedef struct t__instruction_pointer_union
   {
       void *vvv;                          /* to cast to block_if_type, etc. */
       relative_address_type reladdr;      /* relative address of code */
   } instruction_pointer_union_type;

typedef struct t__instruction
   {
       instruction_pointer_union_type ptr;
       opcode_type opcode;          /* instruction operation code */
   } instruction_type;              /*    for_loop_type, assert_type, ... */

typedef struct t__identifier_info
  {
     pointer_union_type ptr;     /* address in memory / function-parm-count */
     union
        {
           struct qqbothidinfqq
              {
                 dim_pair_type first;
                 dim_pair_type second;
              } dims;
           dim_pair_type body;
        } index;
     short scope_level;          /* scope level (negative iff. inactive) */
     char name[IDENT_MAXNCH_P]; /* identifier to search for */
     type_flagword_type flags;  /* type information */
  } identifier_info_type;
```

where the data structure for a dimension pair is of the following type:

```
typedef struct t__dim_pair
  {
    Subscript lower;/* lower dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
    Subscript upper;/* upper dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
  } dim_pair_type;
```

and where the data structure for a pointer union is of the following type:

```
typedef union t__pointer_union
   {
     relative_address_type relative_address;
     short parameter_count;
     void *vvv;                  /* included for completeness */
     Boolean *bbb;               /* used when BOOL_TYPE */
     state_offset_type *sss;     /* used when SSVAR_TYPE */
     char *ccc;                  /* used when CHAR_TYPE */
     int_type *iii;              /* used when INT_TYPE */
     real_type *rrr;             /* used when REAL_TYPE */
   } pointer_union_type;
```

The "ident" field is a pointer into the identifier table. This is the index variable that varies for all values in the set.

The "set_ranges" field is a pointer to an array of set ranges (lower/upper range integer value pairs).

The "set_range_count" field is a count of the number of set ranges in the "set_ranges" array.

The "body" field is a pointer to the beginning of the subroutine that contains the rule-section instructions that fall between the FOR and the ENDFOR and pertain to the current code section.

As an example, consider the following for:

```
(0005): FOR J IN [1..3,7..9]
(0006):     IF NC[10-J] = 0 TRANTO (20,8 OF 0) BY (J*3)*LAMBDA;
(0007): ENDFOR;
```

Figure 22 illustrates the memory layout of the FOR beginning on line 5. This memory layout is also detailed with an excerpt from the memory load map ( **−loadmap** option) in Figure 23.

## 5.9 The FOR index repetition information data structure

During model generation, while executing within the body of a FOR, the index variable is stored in a data structure of the following type:

```
typedef struct t__do_code_stuff
    {
        identifier_info_type *do_idinfo;
        Subscript do_index;
    } do_code_stuff_type;
```

which references the following data types:

```
typedef struct t__identifier_info
  {
```

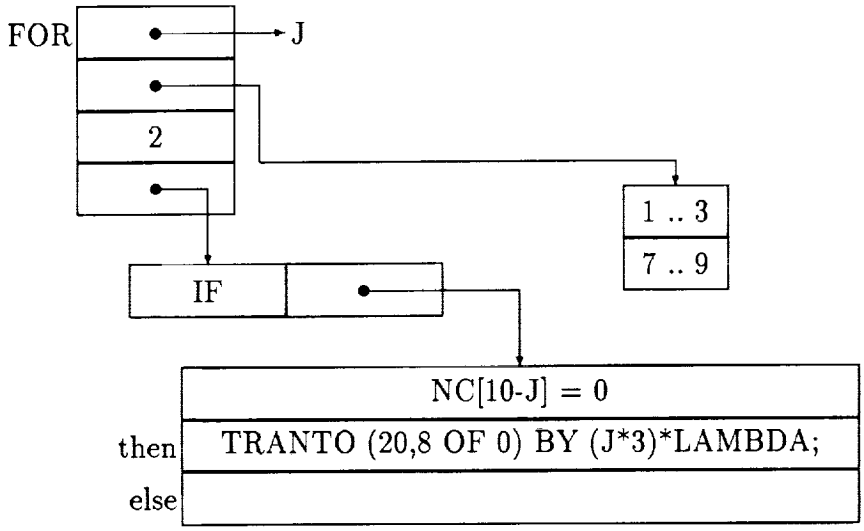



Figure 22: Sample FOR laid out in memory

```
00000218:   [post=(00000597,1),in=(00000598,1),op=(000004D0,1),
             line#=5,err=F,returns(0x03,(<integer>))]
             1
00000230:   [post=(00000599,1),in=(0000059A,1),op=(000004D4,1),
             line#=5,err=F,returns(0x03,(<integer>))]
             3
00000248:   [post=(0000059B,1),in=(0000059C,1),op=(000004D8,1),
             line#=5,err=F,returns(0x03,(<integer>))]
             7
00000260:   [post=(0000059D,1),in=(0000059E,1),op=(000004DC,1),
             line#=5,err=F,returns(0x03,(<integer>))]
             9
00000278:   [post=(0000059F,7),in=(000005A6,8),op=(000004E0,4),
             line#=6,err=F,returns(0x0a,(<boolean>,<expr-var>))]
             NC[10-J]=0
00000290:   [post=(000005AE,1),in=(000005AF,1),op=(000004F0,1),
             line#=6,err=F,returns(0x03,(<integer>))]
             1
...

00000440:   [post=(000005D2,6),in=(000005D8,7),op=(0000055C,3),
             line#=6,err=F,returns(0x0c,(<real>,<expr-var>))]
             (J*3)*LAMBDA

...

000005E4:   (00000218 .. 00000230)   (00000248 .. 00000260)

0000061C:   (TRANTO (exprs=00000290,vars=00000538,#vars=9) BY 1@00000440 (line#6))

...

00000630:   (IF 00000278 THEN GOSUB 000006E8)

0000063C:   (J<16> IN [2@000005E4] GOSUB 00000700)

...


000006E8:   TRANTO 0000061C
000006F4:   RETURN
00000700:   IF 00000630
0000070C:   RETURN

00000718:   LOOP 0000063C
00000724:   RETURN
```

Figure 23: Sample memory map of corresponding FOR

```
    pointer_union_type ptr;     /* address in memory / function-parm-count */
    union
       {
          struct qqbothidinfqq
             {
                dim_pair_type first;
                dim_pair_type second;
             } dims;
          dim_pair_type body;
       } index;
    short scope_level;          /* scope level (negative iff. inactive) */
    char name[IDENT_MAXNCH_P]; /* identifier to search for */
    type_flagword_type flags;  /* type information */
  } identifier_info_type;
```

where the data structure for a dimension pair is of the following type:

```
typedef struct t__dim_pair
  {
    Subscript lower;/* lower dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
    Subscript upper;/* upper dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
  } dim_pair_type;
```

and where the data structure for a pointer union is of the following type:

```
typedef union t__pointer_union
   {
     relative_address_type relative_address;
     short parameter_count;
     void *vvv;                 /* included for completeness */
     Boolean *bbb;              /* used when BOOL_TYPE */
     state_offset_type *sss;    /* used when SSVAR_TYPE */
     char *ccc;                 /* used when CHAR_TYPE */
     int_type *iii;             /* used when INT_TYPE */
     real_type *rrr;            /* used when REAL_TYPE */
   } pointer_union_type;
```

The "do_idinfo" field is a pointer to the identifier table entry for the FOR index variable.

The "do_index" field holds the value of the index variable that is currently in effect.

When a FOR instruction is encountered, the subroutine for the body of the construct is executed for each value in the set of values to use. The index variable and the current value are stored in the "do_stuff_type" data structure and passed to the subroutine evaluator.

## 5.10 The space expression list data structure

When parsing a list of expressions for the destination of a TRANTO, information about the state-space variable that is being modified is stored in an "elist" data structure of the following type:

```
typedef struct t__elist
   {  /* ordering of shorts is important */
      short iii;     /* first.lower */
      short iiiend;  /* first.upper */
      short jjj;     /* second.lower */
      short jjjend;  /* second.upper */
      short knt;
      short *which;
      identifier_info_type *idinfo;
      dim_pair_type q1st; /* first */
      dim_pair_type q2nd; /* second */
      dim_pair_type *q_1st_or_2nd;
      Boolean const1;
      Boolean const2;
      Boolean *qconst;
      Boolean is_var;
   } elist_type;

static elist_type elist;
```

which makes use of additional data structures of the following types:

```
typedef struct t__dim_pair
  {
    Subscript lower;/* lower dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
    Subscript upper;/* upper dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
  } dim_pair_type;

typedef struct t__identifier_info
  {
     pointer_union_type ptr;    /* address in memory / function-parm-count */
     union
        {
           struct qqbothidinfqq
              {
                 dim_pair_type first;
                 dim_pair_type second;
              } dims;
           dim_pair_type body;
        } index;
     short scope_level;         /* scope level (negative iff. inactive) */
     char name[IDENT_MAXNCH_P]; /* identifier to search for */
     type_flagword_type flags;  /* type information */
  } identifier_info_type;
```

where the data structure for a dimension pair is of the following type:

```
typedef struct t__dim_pair
  {
    Subscript lower;/* lower dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
    Subscript upper;/* upper dimension (ARRAY) body-index (FUNCTION/IMPLICIT)*/
  } dim_pair_type;
```

and where the data structure for a pointer union is of the following type:

```
typedef union t__pointer_union
   {
     relative_address_type relative_address;
     short parameter_count;
     void *vvv;                  /* included for completeness */
     Boolean *bbb;               /* used when BOOL_TYPE */
     state_offset_type *sss;     /* used when SSVAR_TYPE */
     char *ccc;                  /* used when CHAR_TYPE */
     int_type *iii;              /* used when INT_TYPE */
     real_type *rrr;             /* used when REAL_TYPE */
   } pointer_union_type;
```

The "iii" field is used for the lower bound of the first subscript of an array. It is used in a loop to check during parse time for an index out of bounds.

The "iiiend" field is used for the upper bound of the first subscript of an array. It is used in a loop to check during parse time for an index out of bounds.

The "jjj" field is used for the lower bound of the second subscript of an array. It is used in a loop to check during parse time for an index out of bounds.

The "jjjend" field is used for the upper bound of the second subscript of an array. It is used in a loop to check during parse time for an index out of bounds.

The "knt" field is used to hold the number of subscripts. The number one is used for a singly subscripted array and the number two is used for a doubly subscripted array. The number SIMPLE_IDENTIFIER is used for non-array state-space variables.

The "which" field is a pointer to either the first ("iii") or second ("jjj") subscript bounds.

The "idinfo" field points into the identifier table to indicate which state-space variable is being updated during the transition.

The "q1st" field is a copy of the dimension pair for the first subscript of the state-space variable being updated during the transition.

The "q2nd" field is a copy of the dimension pair for the second subscript of the state-space variable being updated during the transition.

The "q_1st_or_2nd" field is a pointer to either the "q1st" or the "q2nd" field.

The "const1" field indicates whether the index reference for the first subscript to the state-space variable being updated is a constant expression.

The "const2" field indicates whether the index reference for the second subscript to the state-space variable being updated is a constant expression.

The "qconst" field is a pointer to either the "const1" or the "const2" field.

The "is_var" field indicates that at least one of the first and the second subscript expressions is non-constant.

## 5.11 The built-in function parameter information data structure

When parsing built-in functions, there is a lookup table with information about the quantity, types, and kinds of valid parameters that can be passed to each built-in function in question. Each entry in the table is of the following type:

```
typedef struct t__built_in_parm_info
   {
       short parameter_count;
       type_flagword_type parameter_type;
       type_flagword_type return_type;
       operation_type opcode;
       operation_type aux_opcode;
       char label[16];
   } built_in_parm_info_type;
```

The "parameter_count" field indicates the number of parameters that the built-in function requires. An error message is printed if the number of parameters passed does not equal the number of parameters expected. Certain special values (usually negative numbers to distinguish from actual counts) are allowed. Special values include "VARIAB_LENG", which is used for list functions that can take 1 or more parameters and "ARR_N_IX" for functions that require the name of an array as the first parameter and an index as the second parameter. There are currently no built-in functions that are of length "ARR_N_IX" although the concept applies to some of the Arithmetic/Logic Unit (ALU) operations such as ROWCOUNT, COLCOUNT, ROWSUM, COLSUM, ROWMIN, COLMIN, ROWMAX, COLMAX, ROWANY, COLANY, ROWALL, and COLALL.

The "parameter_type" field indicates the type of parameter(s) required. Unless a type promotion can be made, an error message is printed if the wrong type of a parameter is passed to a built-in function. The ARRAY_TYPE bit is set if the name of an array is legal. If the "parameter_count" is VARIAB_LENG and the ARRAY_TYPE bit is set, the an array is optional. If the "parameter_count" is positive and the ARRAY_TYPE bit is set, then an array is required. If the simple type portion (low three bits) is zero, then the function can accept integers and/or reals.

The "return_type" is in the range 0..7 and gives the simple type of the value that is returned by the function. If set to EMPTY_TYPE, then the "opcode" field applies when all parameters are integers and the "aux_opcode" field applies when at least one parameter is a real. In the case of Boolean functions such as `COUNT`, if set to EMPTY_TYPE, then the "opcode" field applies when more than one parameter is passed and the "aux_opcode" field applies when only one parameter is passed.

The "opcode" field specifies the operation code to use under most circumstances in the postfix expression to evaluate the function result.

The "aux_opcode" field specifies the operation code to use under certain special circumstances in the postfix expression to evaluate the function result.

The "label" field gives the textual name of the built-in function as a convienience for printing out error and warning messages.

## 5.12 The value union data structure

When evaluating postfix expressions, it is necessary to maintain a value stack in addition to an operation stack. Since the values on the stack could be of various types, a union is used for each stack entry. This union is called the "value union" and is of the following type:

```
typedef union t__value_union
   {
     Boolean bbb;              /* used when BOOL_TYPE */
     char ccc;                 /* used when CHAR_TYPE */
     int_type iii;             /* used when INT_TYPE */
     real_type rrr;            /* used when REAL_TYPE */
     state_offset_type sss;    /* used when SSVAR_TYPE */
     pointer_union_type ptr; /* used when ARRAY_TYPE bit is set */
#if defined(INT_32_BIT) || defined(INT_16_BIT)
     struct qqiiis{
        int_type iiia;
        int_type iiib;} pair;
#endif
   } value_union_type;
```

which makes use of additional data structures of the following types:

```
typedef union t__pointer_union
   {
     relative_address_type relative_address;
     short parameter_count;
     void *vvv;                /* included for completeness */
     Boolean *bbb;             /* used when BOOL_TYPE */
     state_offset_type *sss;   /* used when SSVAR_TYPE */
     char *ccc;                /* used when CHAR_TYPE */
     int_type *iii;            /* used when INT_TYPE */
     real_type *rrr;           /* used when REAL_TYPE */
   } pointer_union_type;

typedef short ssvar_value_type;
typedef struct t__state_offset
   {
      ssvar_value_type minval;
      ssvar_value_type maxval;
      bitsize_type bit_offset;
      bitsize_type bit_length;
   } state_offset_type;
```

The "bbb" field applies when Boolean data is in the stack element in question.

The "ccc" field applies when a single character is in the stack element in question. Although the current version of ASSIST does not use character data in expressions, this field is included in the union for completeness.

The "iii" field applies when integer (long) data is in the stack element in question.

The "rrr" field applies when real (double) data is in the stack element in question.

The "sss" field applies when state-space variable offset data is in the stack element in question.

The "ptr" field applies when the starting address of an array is in the stack element in question.

The "pair.iiia" and "pair.iiib" fields are used for efficent copying, pushing, and popping of stack elements. On VAX and SUN systems, two assignment statements execute much faster than a single call to "memcpy". Two versions of the "val_union_cpy" macro are defined in "cm_types.h". One applies only on systems with 16 or 32 bit integers and performs two assignment statements, and the other applies on all other architectures and does a memcpy.

## 5.13 The binary operand pair data structure

Certain data structures are used to hold information about the left and right side of infix/postfix operands as pertaining to the current expression being parsed by the recursive-descent parser. These data structures are:

```
typedef struct t__binary_operand_item_info
   {
      short ixpo;
      short ixin;
      type_flagword_type type;
      type_flagword_type comp;
      type_flagword_type spec;
   } binary_operand_item_info_type;

typedef struct t__binary_operand_pair_info
   {
      binary_operand_item_info_type item[2];
      type_flagword_type ans;
      type_flagword_type spcans;
   } binary_operand_pair_info_type;
```

The "ixpo" field gives the index in the postfix list of the current expression being parsed where the unary/binary operand operation ("V") can be found.

The "ixin" field gives the index in the infix list of the current expression being parsed where the unary/binary operand operation ("V") can be found.

The "type" field gives the type of the operand in question.

The "comp" field gives the computational (or simple) type of the operand in question. The computational type consists of the low three bits of the type.

The "spec" field give the special type of the operand in question. The special type consists of all bits of the type except for the computational bits.

The "item" field is an array containing the above information for each of both the left and right operands of the binary operation being parsed. In the case of a unary operation, the second slot in the array is not used and is not guaranteed to contain any meaningful information.

The "ans" field is used to store the resultant type of the operation being parsed. This starts out as a computational type but is OR'ed with the "spcans" field, depending upon which

operation is being parsed, before being returned to the calling function.

The "spcans" field is used to store the special type bits of the operation being parsed.

## 5.14 The reserved word operator lookup data structure

The reserved word operator lookup table is used by the lexical token scanner to translate from a reserved word that stands for an arithmetic or logical operation to the operation itself. Each entry in the table is of the following type:

```
typedef struct t__rw_operator_lookup
   {
      rwtype rwsrc;
      token tokdest;
   } rw_operator_lookup_type;
```

The "rwsrc" field specifies the scanned reserved word that must be translated from a reserved-word token to an arithmetic or logical operation token.

The "tokdest" field specifies the corresponding token for the translation.

For example, the `RW_AND` reserved word, which is scanned from the word `AND` in the input file, is translated to the token `TK_AND`, which stands for the logical `''&''` operation.

## 5.15 The reserved word lookup data structure

The reserved word lookup table is used by the lexical token scanner to translate from a scanned identifier to its corresponding reserved word. Identifiers that are not in the table are treated as identifiers. Identifiers that are in the table are translated to reserved words. Each entry in the table is of the following type:

```
typedef struct t__reserved_word_lookup
   {
      char text[15];
      rwtype rw;
   } reserved_word_lookup_type;
```

The "text" field contains the name of the identifier that must be translated to a reserved word.

The "rw" field contains the corresponding reserved word to substitute for the identifier during the translation.

For example, the identifier "`TRANTO`" is translated into the reserved word "`RW_TRANTO`".

## 5.16 The token lookup data structure

The token lookup table is used by the lexical token scanner to translate from a symbolic character sequence to its corresponding token. Each entry in the table is of the following type:

```
typedef struct t__token_lookup
   {
     char token[3];
     token value;
   } token_lookup_type;
```

The "token" field gives the text of the symbolic token that must be translated into a single token.

The "value" field give the corresponding token that must be substituted for the contiguous character sequence in the symbolic "token".

For example, the contiguous character sequence "==" is translated into the "TK_BOOL_EQ" token. Also, the contiguous character sequence "<=" is translated into the "TK_LE" token and the sequence "**" is translated into the "TK_POW" token for exponentiation.

## 5.17 The scanning character information data structure

The scanning character information data structure is used to store information about the current character being scanned and the look-ahead character. The ASSIST parser, in most instances, is a single character look-ahead scanner. In some Boolean expressions, the scanner must look ahead to the character following a matching right parenthesis. This is done using "ftell" to remember where in the file the scanner left off and "fseek" to get back after looking ahead.

The data structure is of the following type:

```
typedef struct t__scanning_character_info
   {
        short current_ch_lno;
        short lookahead_ch_lno;
        short current_ch_pos;
        short lookahead_ch_pos;
        char current_ch;
        char lookahead_ch;
   } scanning_character_info_type;
```

The "current_ch_lno" field gives the input file line number on which the current character resides.

The "lookahead_ch_lno" field gives the input file line number on which the lookahead character resides.

The "current_ch_pos" field gives the index in the "current_ch_lno" where the current character resides.

The "lookahead_ch_pos" field gives the index in the "lookahead_ch_lno" where the lookahead character resides.

The "current_ch" field gives the current character itself.

The "lookahead_ch" field gives the lookahead character itself.

## 5.18 Mapping of a program into memory

After an input file has been parsed, ASSIST exits if any syntax errors were detected. In the absence of any syntax errors, the rules and related information are mapped into memory in preparation for model generation.

Memory is allocated and divided into the following sections, which are listed in sequential order:

- Real constants
- Integer constants
- Boolean constants
- State-space variable offsets
- Character strings
- Expressions
- Expression operands
- Expression operations
- Space variable information
- Set range expression pointers
- State-space picture data
- TRANTO clause data
- Block and TRANTO if data
- For range data
- Preamble code
- ASSERT code
- DEATHIF code
- PRUNEIF code

- TRANTO code
- Identifier table

Consider the following example of a complete **ASSIST** input file listing:

```
(0001): {
(0002)X     This system describes an NP-ad (e.g, quintad) with processors which
(0003)X     fail at the rate LAMBDA and recover at the FAST rate DELTA.
(0004)X }
(0005): LAMBDA = 3.0E-4;
(0006): DELTA = 1.0E10;
(0007): NP = 5;
(0008): SPACE = (NWP:0..NP,NFP:0..NP,FAILED:ARRAY[1..NP] OF BOOLEAN);
(0009): START = (NP,0,NP OF FALSE);
(0010): DEATHIF (NFP>=NWP);
(0011): FOR IX IN [1..NP]
(0012):     IF (NWP>0) THEN
(0013):        IF (NOT FAILED[IX]) TRANTO NWP--,NFP++,FAILED[IX]=TRUE BY LAMBDA;
(0014):     ENDIF;
(0015):     IF (NFP>0) THEN
(0016):        IF (FAILED[IX]) TRANTO NWP++,NFP--,FAILED[IX]=FALSE BY FAST DELTA;
(0017):     ENDIF;
(0018): ENDFOR;
(0019): DEATHIF (NFP>=NWP);
```

This example is interesting because, although it is fairly straightforward, it has IF's nested inside of a FOR as well as two IF's in sequence.

Note the inclusion of the redundant `DEATHIF's` on lines 10 and 19. This was intentional in order to illustrate layout in memory due to placement in the input file.

Because memory is mapped using a lot of pointers and symbology, a general description with illustrations will precede the more detailed ones.

The following is a very general synopsis of how the data and code will be laid out in memory:

3.000000000000000e−04
3.000000000000000e−04
1.000000000000000e+10
1.000000000000000e+10

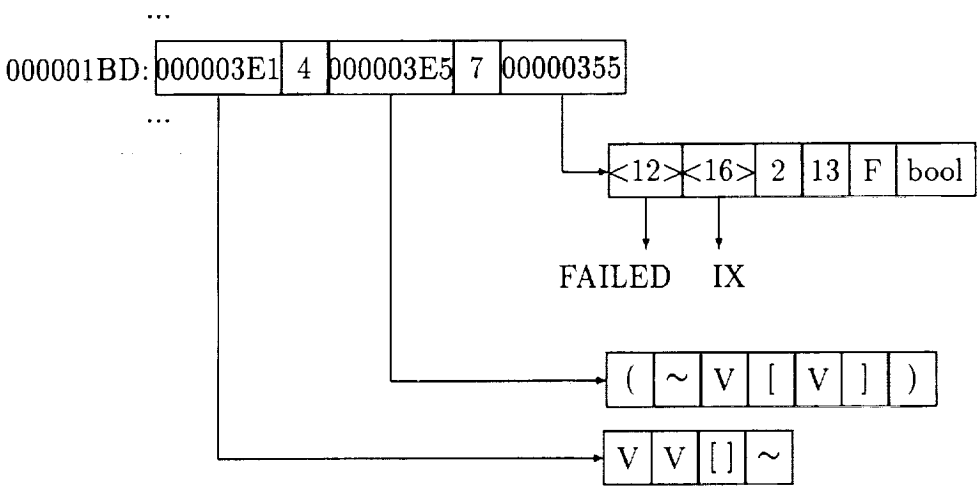
} reals

5
5
0
1
2
3
4
0

} integers

| | |
|---|---|
| FALSE<br>TRUE<br>FALSE<br>FALSE<br>FALSE | Booleans |
| (0..5,3@0)<br>(0..5,3@3)<br>(0..1,1@6) | state offsets |
| NP<br>0<br>1<br>FALSE<br>2<br>FALSE<br>3<br>FALSE<br>4<br>FALSE<br>5<br>FALSE<br>(NFP>=NWP)<br>1<br>NP<br>(NWP>0)<br>(~FAILED[IX])<br>NWP−−<br>NFP++<br>IX<br>TRUE<br>LAMBDA<br>(NFP>0)<br>(FAILED[IX])<br>NWP++<br>NFP−−<br>IX<br>FALSE<br>DELTA<br>(NFP>=NWP) | expressions |

The preceding data is followed by code data structures which is in turn followed by the code itself. In actuality, the expression tokens are not stored in memory in the sequence in which they are printed. There are pointers which point to an infix operation list, a postfix operation list, and and identifier/value operand list. The following diagram shows this in more detail:




The following is a **raw** map of memory after the above example has been parsed:

| | | |
|---|---|---|
| Real constants: | 00000000: | 3.000000000000000e−04 |
| | 00000004: | 3.000000000000000e−04 |
| | 00000010: | 1.000000000000000e+10 |
| | 00000014: | 1.000000000000000e+10 |
| Integer constants: | 00000020: | 5 |
| | 00000024: | 5 |
| | 00000028: | 0 |
| | 0000002C: | 1 |
| | 00000030: | 2 |
| | 00000034: | 3 |
| | 00000038: | 4 |
| | 0000003C: | 0 |
| Boolean constants: | 00000040: | FALSE |
| | 00000041: | TRUE |
| | 00000042: | FALSE |
| | 00000043: | FALSE |
| | 00000044: | FALSE |
| State offset constants: | 00000045: | (0..5,3@0) |
| | 0000004D: | (0..5,3@3) |
| | 00000055: | (0..1,1@6) |

| Expressions: | | | | | | |
|---|---|---|---|---|---|---|
| address | postfix | infix | operand | line | err | returns |
| 0000005D: | (000003B5,1) | (000003B6,1) | (000002F1,1) | 9 | F | <integer> |
| 00000073: | (000003B7,1) | (000003B8,1) | (000002F5,1) | 9 | F | <integer> |
| 00000089: | (000003B9,1) | (000003BA,1) | (000002F9,1) | 9 | F | <integer> |
| 0000009F: | (000003BB,1) | (000003BC,1) | (000002FD,1) | 9 | F | <bool> |
| 000000B5: | (000003BD,1) | (000003BE,1) | (00000301,1) | 9 | F | <integer> |
| 000000CB: | (000003BF,1) | (000003C0,1) | (00000305,1) | 9 | F | <bool> |
| 000000E1: | (000003C1,1) | (000003C2,1) | (00000309,1) | 9 | F | <integer> |
| 000000F7: | (000003C3,1) | (000003C4,1) | (0000030D,1) | 9 | F | <bool> |
| 0000010D: | (000003C5,1) | (000003C6,1) | (00000311,1) | 9 | F | <integer> |
| 00000123: | (000003C7,1) | (000003C8,1) | (00000315,1) | 9 | F | <bool> |
| 00000139: | (000003C9,1) | (000003CA,1) | (00000319,1) | 9 | F | <integer> |
| 0000014F: | (000003CB,1) | (000003CC,1) | (0000031D,1) | 9 | F | <bool> |
| 00000165: | (000003CD,3) | (000003D0,5) | (0000033D,2) | 10 | F | <bool><expr−var> |
| 0000017B: | (000003D5,1) | (000003D6,1) | (00000345,1) | 11 | F | <integer> |
| 00000191: | (000003D7,1) | (000003D8,1) | (00000349,1) | 11 | F | <integer> |
| 000001A7: | (000003D9,3) | (000003DC,5) | (0000034D,2) | 12 | F | <bool>,<expr−var> |
| 000001BD: | (000003E1,4) | (000003E5,7) | (00000355,2) | 13 | F | <bool>,<expr−var> |
| 000001D3: | (000003EC,2) | (000003EE,2) | (0000035D,1) | 13 | F | <null> |
| 000001E9: | (000003F0,2) | (000003F2,2) | (00000361,1) | 13 | F | <null> |
| 000001FF: | (000003F4,1) | (000003F5,1) | (00000365,1) | 13 | F | <null> |
| 00000215: | (000003F6,1) | (000003F7,1) | (00000369,1) | 13 | F | <bool> |
| 0000022B: | (000003F8,1) | (000003F9,1) | (00000379,1) | 13 | F | <real> |
| 00000241: | (000003FA,3) | (000003FD,5) | (0000037D,2) | 15 | F | <bool>,<expr−var> |
| 00000257: | (00000402,3) | (00000405,6) | (00000385,2) | 16 | F | <bool>,<expr−var> |
| 0000026D: | (0000040B,2) | (0000040D,2) | (0000038D,1) | 16 | F | <null> |
| 00000283: | (0000040F,2) | (00000411,2) | (00000391,1) | 16 | F | <null> |
| 00000299: | (00000413,1) | (00000414,1) | (00000395,1) | 16 | F | <null> |
| 000002AF: | (00000415,1) | (00000416,1) | (00000399,1) | 16 | F | <bool> |
| 000002C5: | (00000417,1) | (00000418,1) | (000003A9,1) | 16 | F | <real> |
| 000002DB: | (00000419,3) | (0000041C,5) | (000003AD,2) | 19 | F | <bool>,<expr−var> |

| Expression operands: | | | | | |
|---|---|---|---|---|---|
| address | id | address | id | address | id |
| 000002F1: | <7> | 00000335: | <12> | 00000379: | <3> |
| 000002F5: | <8> | 00000339: | <12> | 0000037D: | <11> |
| 000002F9: | <9> | 0000033D: | <11> | 00000381: | <8> |
| 000002FD: | <0> | 00000341: | <10> | 00000385: | <12> |
| 00000301: | <13> | 00000345: | <9> | 00000389: | <16> |
| 00000305: | <0> | 00000349: | <7> | 0000038D: | <10> |
| 00000309: | <14> | 0000034D: | <10> | 00000391: | <11> |
| 0000030D: | <0> | 00000351: | <8> | 00000395: | <16> |
| 00000311: | <15> | 00000355: | <12> | 00000399: | <0> |
| 00000315: | <0> | 00000359: | <16> | 0000039D: | <10> |
| 00000319: | <6> | 0000035D: | <10> | 000003A1: | <11> |
| 0000031D: | <0> | 00000361: | <11> | 000003A5: | <12> |
| 00000321: | <10> | 00000365: | <16> | 000003A9: | <5> |
| 00000325: | <11> | 00000369: | <1> | 000003AD: | <11> |
| 00000329: | <12> | 0000036D: | <10> | 000003B1: | <10> |
| 0000032D: | <12> | 00000371: | <11> | | |
| 00000331: | <12> | 00000375: | <12> | | |

| Expression operations: | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| address | operations | | | | | | | |
| 000003B5–000003BC: | V | V | V | V | V | V | V | V |
| 000003BD–000003C4: | V | V | V | V | V | V | V | V |
| 000003C5–000003CC: | V | V | V | V | V | V | V | V |
| 000003CD–000003D4: | V | V | >= | ( | V | >= | V | ) |
| 000003D5–000003DC: | V | V | V | V | V | V | > | ( |
| 000003DD–000003E4: | V | > | V | ) | V | V | [] | ~ |
| 000003E5–000003EC: | ( | ~ | V | [ | V | ] | ) | V |
| 000003ED–000003F4: | −− | V | −− | V | ++ | V | ++ | V |
| 000003F5–000003FC: | V | V | V | V | V | V | V | > |
| 000003FD–00000404: | ( | V | > | V | ) | V | V | [] |
| 00000405–0000040C: | ( | V | [ | V | ] | ) | V | ++ |
| 0000040D–00000414: | V | ++ | V | −− | V | −− | V | V |
| 00000415–0000041C: | V | V | V | V | V | V | >= | ( |
| 0000041D–00000420: | V | >= | V | ) | | | | |

| Space variable information: | |
|---|---|
| 00000421: | <10> |
| 00000425: | <11> |
| 00000429: | <12> |

| FOR set range expression pointers: | |
|---|---|
| 0000042D: | (0000017B .. 00000191) |

| state-space PICTURE data: | |
|---|---|
| 00000435: | (NWP:0..5,3@0,NFP:0..5,3@3,FAILED[1..5]:0..1,1@6) |

| ASSERT/DEATHIF/PRUNEIF boolean tests: | |
|---|---|
| address | expression/source line number |
| 0000043F: | (expr=00000165,line#10) |
| 00000445: | (expr=000002DB,line#19) |

| TRANTO clause data structures: | | | | |
|---|---|---|---|---|
| address | (TRANTO (vars,exprs,#vars) | BY | #exprs @ expr | (line#)) |
| 0000044B: | (TRANTO (00000321,0000005D,7) | BY | n/a | (line 9)) |
| 0000045D: | (TRANTO (0000036D,000001D3,3) | BY | 1 @ 0000022B | (line 13)) |
| 0000046F: | (TRANTO (0000039D,0000026D,3) | BY FAST | 1 @ 000002C5 | (line 16)) |

| block and tranto IF data structures: | |
|---|---|
| address | IF ... THEN ... [ELSE ...] |
| 00000481: | (IF 000001BD THEN GOSUB 0000052B) |
| 0000048D: | (IF 000001A7 THEN GOSUB 0000053D) |
| 00000499: | (IF 00000257 THEN GOSUB 0000054F) |
| 000004A5: | (IF 00000241 THEN GOSUB 00000561) |

| FOR range data: | |
|---|---|
| 000004B1: | (<16> IN [1 @ 0000042D] GOSUB 00000573) |

| model generation code, PREAMBLE section: | |
|---|---|
| 000004BF: | BEGIN 000004FE ! ASSERT section |
| 000004C8: | BEGIN 00000507 ! DEATHIF section |
| 000004D1: | BEGIN 00000522 ! PRUNEIF section |
| 000004DA: | BEGIN 0000058E ! TRANTO section |
| 000004E3: | SPACE 00000435 |
| 000004EC: | START 0000044B |
| 000004F5: | END |

| model generation code, ASSERT section: | |
|---|---|
| 000004FE: | RETURN |

| model generation code, DEATHIF section: | |
|---|---|
| 00000507: | DEATHIF 0000043F |
| 00000510: | DEATHIF 00000445 |
| 00000519: | RETURN |

| model generation code, PRUNEIF section: | |
|---|---|
| 00000522: | RETURN |

| model generation code, TRANTO section: | |
|---|---|
| 0000052B: | TRANTO 0000045D |
| 00000534: | RETURN |
| 0000053D: | IF 00000481 |
| 00000546: | RETURN |
| 0000054F: | TRANTO 0000046F |
| 00000558: | RETURN |
| 00000561: | IF 00000499 |
| 0000056A: | RETURN |
| 00000573: | IF 0000048D |
| 0000057C: | IF 000004A5 |
| 00000585: | RETURN |
| 0000058E: | LOOP 000004B1 |
| 00000597: | RETURN |

| Identifier table: | | |
|---|---|---|
| <0> | = | (00000040,SCALAR,0,"FALSE",0x02,[<boolean>]) |
| <1> | = | (00000041,SCALAR,0,"TRUE",0x02,[<boolean>]) |
| <2> | = | (00000000,SCALAR,0,"#3.0E−4",0x04,[<real>]) |
| <3> | = | (00000008,SCALAR,0,"LAMBDA",0x04,[<real>]) |
| <4> | = | (00000010,SCALAR,0,"#1.0E10",0x04,[<real>]) |
| <5> | = | (00000018,SCALAR,0,"DELTA",0x04,[<real>]) |
| <6> | = | (00000020,SCALAR,0,"#5",0x03,[<integer>]) |
| <7> | = | (00000024,SCALAR,0,"NP",0x03,[<integer>]) |
| <8> | = | (00000028,SCALAR,0,"#0",0x03,[<integer>]) |
| <9> | = | (0000002C,SCALAR,0,"#1",0x03,[<integer>]) |
| <10> | = | (00000045,SCALAR,0,"NWP",0x23,[<integer>,<ss−var>]) |
| <11> | = | (0000004D,SCALAR,0,"NFP",0x23,[<integer>,<ss−var>]) |
| <12> | = | (00000055,ARRAY[1..5],0,"FAILED",0xa2,[<boolean>,<ss−var>,<array>]) |
| <13> | = | (00000030,SCALAR,0,"#2",0x03,[<integer>]) |
| <14> | = | (00000034,SCALAR,0,"#3",0x03,[<integer>]) |
| <15> | = | (00000038,SCALAR,0,"#4",0x03,[<integer>]) |
| <16> | = | (0000003C,SCALAR,−1,"IX",0x03,[<integer>]) |

# 6 Hashing of state space

The ASSIST rule generation algorithm uses a hashing algorithm to hash from a given state node n-tuple to the model file state number. Before commencement of rule generation, memory is allocated for the hash table and the state storage array.

The hash table is divided into two sections called the main table and the extension table. The height of the main table is always static because it is the size of the hash table. Both tables are made up of buckets. Each bucket is **bucket_width** wide as defined with the **−bw** option. By default, **−bw=5** . In case more than **bucket_width** states map to the same bucket, there is a link at the end of the bucket which points to the next bucket in the linked list. The extension table is initially a free bucket pool. When a new bucket is needed to extend the main table, the new bucket is always taken from the free pool. When the free pool becomes empty and a new bucket is required, an attempt is make to re-allocate a larger extension table. On MS-DOS systems, all available memory is allocated for the extension bucket. On all systems, an error message is printed out when there is no available memory left for the re-allocation.

The state storage array holds the bit-encoded state-space nodes. The index into this array is the state number less some constant. There are some extra special states which are stored at the front of the table. The death and prune states are omitted unless **ONEDEATH** is **OFF** in which case only the prune states are omitted and the included death states have an extra death-state flag bit set.

The hash table is packed into character arrays:

```
static unsigned char *state_storage;
static unsigned char *bucket_storage;
static unsigned char *bucket_extension_storage;
static unsigned char *bucket_extension_ovf;
static unsigned char *next_free_extension_bucket;
```

The "state_storage" array holds the bit-packed state nodes for each of the states in the model. The index into this array is computed as follows:

$$i = x + h - s$$

where $i$ is the index into the array, $x$ is the state number output to the model file, $h$ is the number of special state node entries in header, and $s$ is the state number of the start state. To get the byte index:

$$i_b = i \times w$$

where $w$ is the number of bytes necessary to pack a state node. See Section 4.8 on page 28 for detailed information on how state nodes are packed.

The "bucket_storage" array is used to store the main hash table and the "bucket_extension_storage" array is used to store the extension links for the entries in the main hash table that have more than "bucket_width" collisions.

In order to conserve memory, the values stored in the buckets are packed as three-byte integers. A three byte integer can store numbers in the range -8,388,607 through 8,388,607. For the purpose of illustration, the fictitious "C" language type "medium" will be used to denote a three byte integer:

```
typedef unsigned char medium[3];  /* three byte integer */
```

The main hash table array is sub-divided as if there was a type as follows:

```
typedef struct t__main_table_bucket
   {
       medium count;                  /* count of entries */
       medium nextlink;               /* link to next bucket */
       medium entry[bucket_width];    /* each state number */
   } main_table_bucket_type;
```

Note that the "count" is the cumulative total of all entries in all buckets. If, for example, the count is 11 and the width is 5, then there are three buckets in the chain. The first two buckets, having 5 entries, will be full. The third bucket will have the remaining entry. The first bucket will be in the main table and the remaining buckets will be found in the extension table. The extension hash table array is sub-divided as if there was a type as follows:

```
typedef struct t__extension_table_bucket
   {
       medium nextlink;               /* link to next bucket */
       medium entry[bucket_width];    /* each state number */
   } extension_table_bucket_type;
```

Consider a system with a combined total of 6 `DEATHIF` and `PRUNEIF` statements. Then the start state will be state 7. Suppose that states 67 and 103 hash to bucket #1, no states hash to bucket #2, the start state number 7 hashes to bucket #3, six states numbers 73, 82, 91, 101, 104, and 122 hash to bucket $\#n-1$, and that state number 197 hashes to the last bucket. The diagram in figure 24 illustrates, for a bucket width of 5, how these collisions are hashed into the buckets.

Note that the link is drawn to the right side of each bucket even though it is physically located as the second "medium" in the main table bucket and the first "medium" in the extension table bucket. This makes the arrows easier to draw and makes the picture less cluttered.

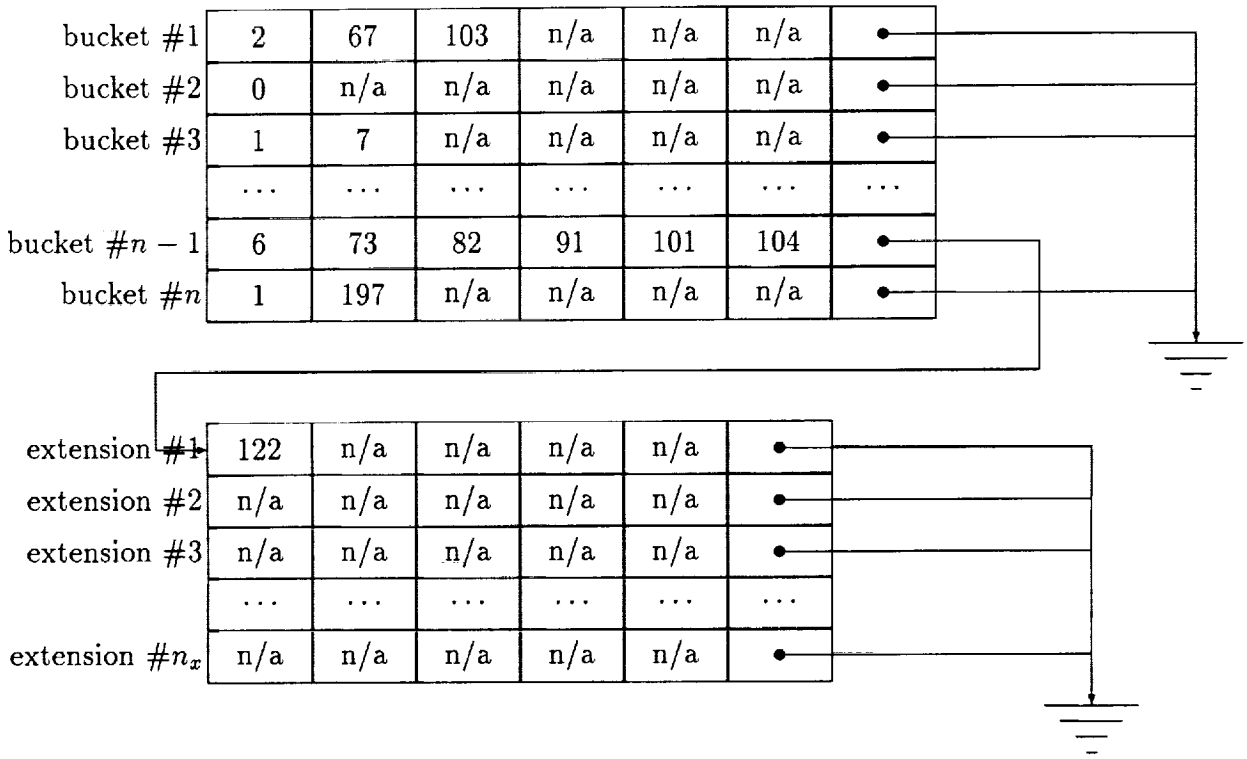



Figure 24: Sample hash table laid out in memory

# 7 Concluding Remarks

The internal data structures and algorithms of the ASSIST program have been explained in detail. The ASSIST program can form a convenient base for the development of interface programs to provide the user with interface capabilities of different forms or higher levels of abstraction, as was done with TOTAL prototype[2, 3] and the ARM program[4]. Reliability model solvers can be tied directly to the ASSIST internal language, as was done with the ASSURE research prototype[5], obviating the need for writing the huge generated model to a file. Through these projects, ASSIST has demonstrated its usefulness as a general platform for reliability analysis tool development, and we would welcome other researchers' use of the program in a similar manner. This is the reason that we undertook the effort to document the internals of the program to the level of detail given in this manual.

# References


[1] Johnson, Sally C. and Boerschlein, David P.: *ASSIST User Manual.* NASA Technical Memorandum 4592, 1994.

[2] Johnson, Sally C. and Butler, Ricky W.: *A Table Oriented Interface for Reliability Modeling of Fault-Tolerant Architectures*, Proceedings of the IEEE/AIAA $11^{th}$ Digital Avionics Systems Conference, Seattle, Washington, October 1992.

[3] Johnson, Sally C. and Boerschlein, David P., *TOTAL User Manual*, NASA Technical Memorandum 109101, 1994.

[4] Liceaga, Carlos A. and Siewiorek, Daniel P.: *Automatic Specification of Reliability Models for Fault-Tolerant Computers*, NASA Technical Paper 3301, July 1993.

[5] Daniel L. Palumbo: *Using Failure Modes and Effects Simulation as a Means of Reliability Analysis*, Proceedings of the IEEE/AIAA $11^{th}$ Digital Avionics Systems Conference, Seattle, Washington, October 1992.

# A BNF Language Description

This appendix gives a complete description of the ASSIST language syntax using the "Backus-Naur Form" grammar.

```
<program>          ::= <setup-section>  <start-section>  <rule-section>

<setup-section>    ::= <setup-stat-seq>  <SPACE-stat>

<start-section>    ::= <start-stat-seq>  <START-stat>  <start-stat-seq>

<rule-section>     ::= <rule-stat-seq>


<setup-stat-seq>   ::= ε
                     | <any-setup-sec-stat>  <setup-stat-seq>

<start-stat-seq>   ::= ε
                     | <any-start-sec-stat>  <start-stat-seq>

<rule-stat-seq>    ::= <any-rule-sec-stat>
                     | <any-rule-sec-stat>  <rule-stat-seq>


<any-setup-sec-stat>  ::= <global-stat>
                        | <pre-rule-global-stat>

<any-start-sec-stat>  ::= <global-stat>
                        | <pre-rule-global-stat>
                        | <dep-variable-def>
                        | <function-def>
                        | <impl-function-def>

<any-rule-sec-stat>   ::= <global-stat>
                        | <ASSERT-stat>
                        | <DEATHIF-stat>
                        | <PRUNEIF-stat>
                        | <TRANTO-stat>
                        | <IF-stat>
                        | <FOR-stat>
```

<pre-rule-global-stat> ::= <quoted-SURE-stat>
| <constant-def-stat>
| <option-def-stat>
| <INPUT-stat>

<global-stat> ::= <debug-stat>
| <command-option-stat>
| <empty-stat>

<any-statement> ::= <any-setup-sec-stat>
| <any-start-sec-stat>
| <any-rule-sec-stat>
| <SPACE-stat>
| <START-stat>

<reserved-word> ::= <sensitive-keyword>
| <built-in-func-name>
| <pre-defined-constant>
| <descriptive-operator>
| <statement-name>

<sensitive-keyword> ::= **BY**
| **FAST**
| **THEN**
| **ELSE**
| **ENDIF**
| **ENDFOR**
| **WITH**
| **OF**
| **IN**
| **ARRAY**
| **ON**
| **OFF**
| **FULL**
| **BOOLEAN**

<pre-defined-constant> ::= <option-def-name>
| **AUTOFAST**
| **TRIMOMEGA**
| **TRUE**
| **FALSE**

```
<descriptive-operator> ::= AND
                         | OR
                         | NOT
                         | MOD
                         | CYC
                         | DIV

<statement-name> ::= <option-def-name>
                   | C_OPTION
                   | DEBUG$
                   | INPUT
                   | SPACE
                   | FUNCTION
                   | IMPLICIT
                   | VARIABLE
                   | START
                   | ASSERT
                   | DEATHIF
                   | PRUNEIF
                   | PRUNIF
                   | TRANTO
                   | IF
                   | FOR

<option-def-name> ::= ONEDEATH
                    | COMMENT
                    | ECHO
                    | TRIM
```

```
<constant-def-stat> ::= <named-constant> = <const-var-def-clause> ;
<const-var-def-clause> ::= <constant-def-clause>
                         | BOOLEAN <constant-def-clause>

<constant-def-clause> ::= <expr> ;
                        | ARRAY ( <expr-list-with-of> ) ;
                        | <single-sub-array> ;
                        | <double-sub-array> ;

<double-sub-array> ::= [ <sub-array-list> ]

<sub-array-list> ::= <single-sub-array> , <single-sub-array>
                   | <single-sub-array> , <sub-array-list>

<single-sub-array> ::= [ <expr-list-with-of> ]
```

```
<option-def-stat> ::= ONEDEATH <flag-status> ;
                    | COMMENT <flag-status> ;
                    | ECHO <flag-status> ;
                    | TRIM <flag-status> ;
                    | TRIM <flag-status> WITH <expr> ;

<INPUT-stat> ::= INPUT <input-list> ;

<SPACE-stat> ::= SPACE = <space-picture> ;


<function-def> ::= FUNCTION <function-name> ( <function-parm-list> ) = <expr> ;

<impl-function-def> ::= IMPLICIT <impl-func-name> [ <impl-parm-list> ] = <expr> ;
                      | IMPLICIT <impl-func-name>
                                 [ <impl-parm-list> ] ( <index-parm-list> ) = <expr> ;

<dep-variable-def> ::= VARIABLE <impl-func-name> [ <impl-parm-list> ] = <const-var-def-clause> ;


<START-stat> ::= START = <space-expression> ;


<ASSERT-stat> ::= ASSERT <boolean-expression> ;

<DEATHIF-stat> ::= DEATHIF <boolean-expression> ;

<PRUNEIF-stat> ::= PRUNEIF <boolean-expression> ;
                 | PRUNIF <boolean-expression> ;

<TRANTO-stat> ::= IF <boolean-expression> <TRANTO-clause> ;
                | <TRANTO-clause> ;

<IF-stat> ::= IF <boolean-expression> THEN
                 <rule-stat-seq>
              ENDIF ;
            | IF <boolean-expression> THEN
                 <rule-stat-seq>
              ELSE
                 <rule-stat-seq>
              ENDIF ;

<FOR-stat> ::= FOR <for-range>
                   <rule-stat-seq>
               ENDFOR ;
```

```
<quoted-SURE-stat>  ::=  " <quot-text> "

<command-option-stat>  ::=  C_OPTION <identifier> ;
                       |    C_OPTION <identifier> = <value> ;

<debug-stat>  ::=  DEBUG$ ;
              |    DEBUG$ <identifier> ;
<empty-stat>  ::=  ;

<TRANTO-clause>  ::=  TRANTO <space-destination-list> BY <rate-expression>
                 |    TRANTO <space-expression> BY <rate-expression>

<flag-status>  ::=  ε
               |    OFF
               |    ON
               |    FULL
               |    = 0
               |    = 1
               |    = 2

<input-list>  ::=  <input-item>
              |    <input-item> , <input-list>

<input-item>  ::=  <named-constant>
              |    <prompt-message> : <named-constant>
              |    BOOLEAN <named-constant>
              |    BOOLEAN <prompt-message> : <named-constant>

<prompt-message>  ::=  " <quot-text> "

<function-parm-list>  ::=  ε
                      |    <identifier>
                      |    <identifier> , <function-parm-list>

<index-parm-list>  ::=  <identifier>
                   |    <identifier> , <index-parm-list>
```

<impl-parm-list> ::= <state-space-var>
| <state-space-var> , <impl-parm-list>

<quot-text> ::= $\varepsilon$
| <quot-text-char> <quot-text>

<quot-text-char> ::= <non-quote-ascii-char>

<space-expression> ::= ( <space-expr-list> )

<space-expr-list> ::= <space-expr-item>
| <space-expr-item> , <space-expr-list>

<space-expr-item> ::= <whole-or-boolean-expression>
| <whole-expression> **OF** <whole-or-boolean-expression>
| <space-expression>

<space-picture> ::= ( <space-item-list> )

<space-item-list> ::= <space-item>
| <space-item> , <space-item-list>

<space-item> ::= <state-space-var>
| <state-space-var> : <i-range>
| <state-space-var> : **BOOLEAN**
| <space-picture>
| <state-space-var> : **ARRAY** [ <array-range> ]
| <state-space-var> : **ARRAY** [ <array-range> ] **OF** <i-range>
| <state-space-var> : **ARRAY** [ <array-range> ] **OF BOOLEAN**

<array-range> ::= <i-range>
| <i-range> , <i-range>

<space-destination-list> ::= <space-destination>
| <space-destination> , <space-destination-list>

<space-destination> ::= <dest-adj-clause>

<dest-adj-clause> ::= <state-space-var> = <whole-or-boolean-expression>
| <state-space-var> <inc-op>

<for-range> ::= <index-variable> = <whole-expression> , <whole-expression> †
| <index-variable> **IN** <set>

<set> ::= [ <set-range-list> ]

<set-range-list> ::= <i-range>
| <whole-expression>
| <i-range> , <i-range-list>

<i-range> ::= <lower-bound> .. <upper-bound>

<lower-bound> ::= <range-bound>

<upper-bound> ::= <range-bound>

<range-bound> ::= <whole-expression> §

<rate-expression> ::= <real-expression>
| < <real-expression> , <real-expression> >
| < <real-expression> , <real-expression> , <real-expression> >
| **FAST** <real-expression>

<expr-list-with-of> ::= <expr>
| <whole-expression> **OF** <expr>
| <expr> , <expr-list-with-of>
| <whole-expression> **OF** <expr> , <expr-list-with-of>

<expression-list> ::= <expr>
| <expr> , <expression-list>

<built-in-expr-list> ::= <expr>
| <expr> , <built-in-expr-list>
| <wild-sub-array>
| <wild-sub-array> , <built-in-expr-list>

```
<wild-sub-array>  ::=  <named-constant>  [  *  ,  <whole-expression>  ]
                   |  <named-constant>  [  <whole-expression>  ,  *  ]
                   |  <state-space-var>  [  *  ,  <whole-expression>  ]
                   |  <state-space-var>  [  <whole-expression>  ,  *  ]

<expr>  ::=  <real-expression>
          |  <whole-expression>
          |  <boolean-expression>

<whole-or-boolean-expression>  ::=  <whole-expression>
                                 |  <boolean-expression>

<whole-expression>  ::=  <integer-expression>

<real-expression>  ::=  <numeric-expression>

<integer-expression>  ::=  <numeric-expression>
```

```
<boolean-expression>  ::=  <bool-term-expr>

<bool-term-expr>  ::=  <bool-term>
                   |   <bool-term-expr>  <or-op>  <bool-term>

<bool-term>  ::=  <bool-factor>
              |   <bool-term>  <and-op>  <bool-factor>

<bool-factor>  ::=  <bool-item>
                |   <bool-item>  ==  <bool-item>

<bool-item>  ::=  <numeric-comparison>
              |   <simple-bool-item>

<numeric-comparison>  ::=  <whole-expression>  <relation>  <whole-expression>

<simple-bool-item>  ::=  <non-index-single-item>
                     |   <truth-value>
                     |   <boolean-function-invocation>
                     |   ( <boolean-expression> )
                     |   NOT  <simple-bool-item>

<or-op>  ::=  OR
           |  |
           |  XOR

<and-op>  ::=  AND
            |  &

<relation>  ::=  <inequality-relation>
             |   <equality-relation>

<inequality-relation>  ::=  >
                        |   <
                        |   >=
                        |   <=

<equality-relation>  ::=  <>
                      |   =
```

```
<numeric-expression>  ::=  <term-expr>

        <term-expr>   ::=  <term>
                      |    <term-expr>   <add-op>   <term>

            <term>    ::=  <factor>
                      |    <term>   <mpy-op>   <factor>

          <factor>    ::=  <numeric-item>
                      |    <numeric-item>   <pow-op>   <factor>

    <numeric-item>    ::=  <bin-numeric-item>
                      |    <sign-op>   <numeric-item>

<bin-numeric-item>    ::=  <non-index-single-item>
                      |    <index-variable>
                      |    <unsigned-value>
                      |    <named-constant>   <cat-op>   <bin-numeric-item>
                      |    <numeric-function-invocation>
                      |    (  <numeric-expression>  )


         <add-op>     ::=  +
                      |    −

         <mpy-op>     ::=  *
                      |    /
                      |    MOD
                      |    CYC
                      |    DIV

         <pow-op>     ::=  **

         <sign-op>    ::=  −

          <inc-op>    ::=  ++
                      |    −−

          <cat-op>    ::=  ∧
```

```
<boolean-function-invocation>  ::=  <function-invocation>

<numeric-function-invocation>  ::=  <function-invocation>

      <function-invocation>    ::=  <impl-func-name>
                                  |  <function-name>  (  <expression-list>  )
                                  |  <built-in-name>  (  <built-in-expr-list>  )

   <non-index-single-item>     ::=  <named-constant>
                                  |  <named-constant>   [ <whole-expression> ]
                                  |  <named-constant>   [ <whole-expression> , <whole-expression> ]
                                  |  <state-space-var>
                                  |  <state-space-var>  [ <whole-expression> ]
                                  |  <state-space-var>  [ <whole-expression> , <whole-expression> ]

        <function-name>           ::=  <identifier>

        <impl-func-name>          ::=  <identifier>

        <built-in-name>           ::=  SQRT     |  EXP      |  LN
                                  |  SIN      |  COS      |  TAN
                                  |  ARCSIN   |  ARCCOS   |  ARCTAN
                                  |  FACT     |  SUM      |  COUNT
                                  |  COMB     |  PERM     |  ABS
                                  |  ANY      |  ALL      |  SIZE
                                  |  MIN      |  MAX

        <truth-value>             ::=  FALSE
                                  |  TRUE

        <comment>                 ::=  (*  <text>  *)
                                  |  {  <text>  }

        <under>                   ::=  _

        <dollar>                  ::=  $

        <E-char>                  ::=  E  | e  | D  | d
```

```
<letter>  ::=  A | B | C | D | E
               | F | G | H | I | J
               | K | L | M | N | O
               | P | Q | R | S | T
               | U | V | W | X | Y
               | Z
               | a | b | c | d | e
               | f | g | h | i | j
               | k | l | m | n | o
               | p | q | r | s | t
               | u | v | w | x | y
               | z

<digit>  ::=  0 | 1 | 2 | 3 | 4
              | 5 | 6 | 7 | 8 | 9

<ident-char>  ::=  <letter>
                   | <digit>
                   | <under>
                   | <dollar> ‡

<identifier>  ::=  <letter>
                   | <letter>  <ident-rest>

<ident-rest>  ::=  <ident-char>
                   | <ident-char>  <ident-rest>

<unsigned-integer-value>  ::=  <digit>
                               | <digit>  <unsigned-integer-value>

<unsigned-real-value>  ::=  <unsigned-integer-value> . <unsigned-integer-value>
                            | <unsigned-integer-value> . <unsigned-integer-value>  <exponent-value>

<exponent-value>  ::=  <E-char>  <sign-op>  <unsigned-integer-value>
                       | <E-char>  <unsigned-integer-value>

<named-constant>  ::=  <identifier>

<state-space-var>  ::=  <identifier>

<index-variable>  ::=  <identifier>
```

‡ The C_OPTION statement can be used to disable all or individual warning messages.

‡ All identifiers with dollar signs are reserved by the ASSIST language.

§ Although lower and upper bounds can take on values between 0 and 32767, their difference must be no more than 255.

# B Command Line Options

The ASSIST command line allows the user to specify options. These options control a number of parameters and allow the user more control over how the ASSIST program executes.

Options must be preceded by a slash under VMS as in:

**/map**

and must be preceded by a dash under UNIX as in:

**−map**

Options may be specified either in upper or lower case. The normal UNIX case sensitivity does not apply to the ASSIST command line options.

Options may also be typed into the input file via `C_OPTION` commands. These commands must precede all other commands including any other debug commands. For example, the statement:

```
C_OPTION LEL=10;
```

in the input file is the same as the following command-line options:

**/lel=10**

or

**−lel=10**

The following options are available:

- **−c** ⟶ Specifies identifier case sensitivity. Use of " **−c** " is not recommended since SURE is never case sensitive. Case sensitive state-space variables are safe to use because they are never passed to SURE. Case sensitive constant names will cause problems because they are passed to SURE. The default is no case sensitive identifier names.
- **−pipe** ⟶ This option causes the model output to be written to standard output instead of to a model file. It is useful if one wishes to pipe the model directly to SURE. This option is valid only under UNIX. An attempt to use it under VMS will cause ASSIST to print an error message. The default is no rerouting of the model file to the standard output file.
- **−map** ⟶ This option causes ASSIST to produce a cross reference map of all of the definitions of and references to identifiers and literal values in the program. The map also tells which **ENDFOR** matches which **FOR** and which **ENDIF** matches which **IF**. It also indicates to which **IF** an **ELSE** belongs. Although the map is several pages long, it may help the user to find misspelled identifiers. Its use is recommended during the first few executions of a new input file. The default is no cross reference map.
- **−xref** ⟶ This option is the same as the **−map** option.
- **−loadmap** ⟶ This option is used to request a load map of the internal data structures and memory allocation generated during the parsing of the input file. The information produced is extremely technical. The option remains in the language for verification purposes and because it is useful under some rare instances. Its use is **not** recommended. Use of **−xref** is recommended instead. The default is no load map.

- **−ss** ⟶ This option forces ASSIST to print the level of each warning as part of the warning message. For example, instead of `[WARNING]`, the message will read `[WARNING SEVERITY 3]` The default is no display of warning severity.
- **−we3** ⟶ This option forces ASSIST to abbreviate to three letters in warning and error messages as in `[ERR]` and `[WRN]` The default is no abbreviation of the words "ERROR" and "WARNING".
- **−bat** ⟶ This option causes ASSIST to execute in batch mode. In batch mode, the command line is echoed to standard error (usually the user's monitor screen). The default is no echoing of the command line used to invoke ASSIST.
- **−wid** =*nnn* ⟶ This option specifies the overflow length of a line. The default is 80 characters, which results in an effective input line length of 79 characters.
- **−tab** =*nnn* ⟶ This option specifies how many spaces are equivalent to a tab character. The default is four spaces per tab.
- **−nest** =*nnn* ⟶ Specifies how deeply a space statement can be recursively nested. The default is 16 on most systems (8 on the IBM PC).
- **−rule** =*nnn* ⟶ Specifies the maximum number of rules that can be nested inside a single block `IF` or `FOR` construct. The default is 4096 for most systems (1024 on the IBM PC).
- **−pic** =*nnn* ⟶ Specifies the maximum number of nodes that can be on the stack when parsing a state-space picture. The number of state-space variables may exceed this number only if the state-space picture is recursively defined. The default is 100.
- **−lel** =*nnn* ⟶ Specifies the "line error limit". If the number of errors per line ever exceeds this value, then ASSIST will quit processing the input file immediately after printing one additional and appropriate error message. The default is a maximum of 5 errors allowed per line.
- **−lwl** =*nnn* ⟶ Specifies the "line warning limit". If the number of warnings per line ever exceeds this value, then ASSIST will quit processing the input file immediately after printing an appropriate error message. The default is a maximum of 5 warnings allowed per line.
- **−el** =*nnn* ⟶ Specifies the "error limit". If the cumulative number of errors ever exceeds this value, then ASSIST will quit processing the input file immediately after printing one additional and appropriate error message. The default is a maximum of 40 errors allowed per input file.
- **−wl** =*nnn* ⟶ Specifies the "warning limit". If the cumulative number of warnings ever exceeds this value, then ASSIST will quit processing the input file immediately after printing an appropriate error message. The default is a maximum of 40 warnings allowed per input file.
- **−bc** =*nnn* ⟶ Specifies the "bucket count" for the rule generation state hashing algorithm. If rule generation is taking a long time because of identifier hash clashes, then this value can be adjusted. The default bucket count is 1009.
- **−bi** =*nnn* ⟶ Specifies the "bucket increment". This controls how many additional state buckets will be allocated at a time when the system runs out of buckets.
- **−bw** =*nnn* ⟶ Specifies the "bucket width" (i.e., the number of states that will fit in a single link of the linked list for each bucket) for the rule generation state hashing algorithm. If rule generation is taking a long time because of identifier hash clashes, then this value can be adjusted. The default bucket width is 5.
- **−lp** =*nnn* ⟶ Specifies the number of lines per page on the log file. The default is 58 lines maximum per page on the log file.
- **−i** =*nnn* ⟶ Specifies the maximum number of identifiers that can be held in the identifier table. The default is a maximum of 400 unique identifier names in the table for most systems (200 on the IBM PC).

- **−n** =*nnn* ⟶ Specifies the maximum number of literal values that can be held in the identifier table. Note that "6.0" and "6.00" are considered as two different entries in the table so that they can be written to the model file the same way they were typed into the input file. The default is a maximum of 200 unique numerical values in the table for most systems (50 on the IBM PC).
- **−o** =*nnn* ⟶ Specifies the maximum number of operands that can be held in the expression operand list while parsing a single statement. The default is 300 on most systems (50 on the IBM PC). The maximum number of infix/postfix operations is a function of this number and is always significantly greater.
- **−e** =*nnn* ⟶ Specifies the maximum number of expressions that can be held while parsing a single statement. The default is 300 on most systems (50 on the IBM PC).
- **−p** =*nnn* ⟶ Specifies the maximum number of identifiers for a FUNCTION or IMPLICIT or VARIABLE parameter list. The default is 64 on most systems (32 on the IBM PC).
- **−b** =*nnn* ⟶ Specifies the maximum number of tokens in the body per FUNCTION or IMPLICIT or VARIABLE definition. The default is 1024 on most systems (256 on the IBM PC).
- **−w** =*nnn* ⟶ Specifies the levels of warnings that will be issued. The higher the number, the more warnings. Levels available are 0 for no warnings through 99 for all warnings. There are currently only three levels defined. The default is two levels of warning reporting. The **W=FEWER** form decreases the level to one less level of warnings. The **W=NONE** form suppresses all warnings. The **W=ALL** form enables all warnings.

# C Notes for VMS Users

Certain peculiarities of the VMS operating system are described in this appendix. The VMS user should keep these in mind when executing ASSIST in order to save time and frustration.

## C.1 Special VMS Errors

The first thing to note about VMS is that there are certain errors that the VMS operating system detects that standard UNIX "C" does not. These errors do not have "C" error numbers. Since ASSIST is written in "C" it was decided that, when one of these special error numbers would arise, the number of the error would be printed. For example, consider the following screen session from **ASSIST**:

```
[ERROR] SPECIAL VMS ERROR NUMBER: 100052
[ERROR] QUITTING COMPILATION !!!

0002 ERRORS.
$
```

To check the meaning of the error number 100052, use the VMS **exit** command as illustrated in the following example:

```
$ exit 100052
%RMS-F-SYN, file specification syntax error
$
```

## C.2 Model Cannot be Piped

The next thing to note about VMS is that, unlike UNIX and IBM MS DOS, the standard output from one command cannot be piped to the standard input of another command. The **/pipe** option is therefore disallowed on VMS systems.

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

| 1. AGENCY USE ONLY *(Leave blank)* | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | April 1994 | Technical Memorandum |

**4. TITLE AND SUBTITLE**
ASSIST Internals Reference Manual

**5. FUNDING NUMBERS**
WU 505-64-10-07

**6. AUTHOR(S)**
Sally C. Johnson
David P. Boerschlein

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
NASA Langley Research Center
Hampton, VA 23681-0001

**8. PERFORMING ORGANIZATION REPORT NUMBER**

**9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
National Aeronautics and Space Administration
Washington, DC 20546-0001

**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
NASA TM-109102

**11. SUPPLEMENTARY NOTES**
Sally C. Johnson: Langley Research Center, Hampton, VA
David P. Boerschlein: Lockheed Engineering & Sciences Company, Hampton, VA

**12a. DISTRIBUTION / AVAILABILITY STATEMENT**
Unclassified-Unlimited

Subject Category 62

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** ***(Maximum 200 words)***

The Abstract Semi-Markov Specification Interface to the SURE Tool (ASSIST) program was developed at NASA Langley Research Center in order to analyze the reliability of virtually any fault-tolerant system. A user manual was developed to detail its use. Certain technical specifics are of no concern to the end user, yet are of importance to those who must maintain and/or verify the correctness of the tool. This document takes a detailed look into these technical issues.

**14. SUBJECT TERMS**
Reliability Modeling, Reliability Analysis, Fault Tolerance

**15. NUMBER OF PAGES**
101

**16. PRICE CODE**
A06

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| Unclassified | Unclassified | | |